Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Marko Škrjanec

Zasnova visoko skalabilne arhitekture za razvoj lokacijsko odvisnih vmesnikov API v oblaku

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2016

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Marko Škrjanec sem avtor magistrskega dela z naslovom:

Zasnova visoko skalabilne arhitekture za razvoj lokacijsko odvisnih vmesnikov API v oblaku

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaž Branko Jurič,
- so elektronska oblika magistrskega dela, naslov (slovenski, angleški), povzetek (slovenski, angleški) ter ključne besede (slovenske, angleške) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 1. september 2016 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču za vso strokovno pomoč in napotke pri izdelavi magistrske naloge. Iskreno se zahvaljujem tudi staršem, Barbari, prijateljem, članom laboratorija in sodelavcem za vse vesele trenutke, pomoč in podporo tekom celotnega študija.

Marko Škrjanec, 2016

Kazalo

Seznam uporabljenih kratic in simbolov

API Application Programming Interface (aplikacijski programski vmesnik)

CDI Contexts & Dependency Injection (dodajanje konteksta in vključevanje odvisnosti platforme Java)

DB Database (podatkovna zbirka)

DRY Don't Repeat Yourself (princip o ponovni uporabi programske kode)

EAR Enterprise Archive (skupni arhiv za združevanje modulov platforme Java)

EL Expression Language (jezik izrazov platforme Java)

ESB Enterprise Service Bus (komunikacijski kanal storitev)

GPS Global Positioning System (sistem globalnega pozicioniranja)

HTTP Hypertext Transfer Protocol (komunikacijski protokol za prenos nadbesedila)

IaaS Infrastructure as a Service (infrastruktura kot storitev)

IP Internet Protocol (internetni protokol)

Java EE Java Platform, Enterprise Edition (izdaja platforme Java za poslovne informacijske sisteme)

Java ME Java Platform, Micro Edition (mikro izdaja platforme Java)

JAR Java Archive (arhiv platforme Java)

JAX-RS Java API for RESTful Services (vmesnik API za storitve REST platforme Java)

JPA Java Persistence API (vmesnik API platforme Java za upravljanje relacijskih podatkov)

JSON JavaScript Object Notation (berljiva oblika zapisa podatkovnih objektov)

JSF JavaServer Faces (ogrodje za spletne strani platforme Java)

JSP JavaServer Pages (spletne strani platforme Java)

PaaS Platform as a Service (platforma kot storitev)

REST Representational State Transfer (arhitekturni način prenašanja stanj)

RPC Remote Procedure Calls (oddaljeni klici procedur)

SaaS Software as a Service (programska oprema kot storitev)

SOA Service-Oriented Architecture (storitveno usmerjena arhitektura)

SOAP Simple Object Access Protocol (protokol za izmenjavo podatkov)

UI User Interface (uporabniški vmesnik)

URI Uniform Resource Identifier (enoten identifikator virov)

WS Web Services (spletne storitve)

WSDL Web Service Definition Language (jezik za definicijo vmesnikov spletnih storitev)

XML Extensible Markup Language (razširljiv označevalni jezik)

Povzetek

Danes že na dnevni ravni uporabljamo storitve s kontekstno informacijo o lokaciji za izboljšanje uporabniške izkušnje. Lokacijske storitve so dobro opredeljene in zanje obstaja več arhitektur, predvsem v obliki monolitnih rešitev. Težava obstoječih arhitektur je v njihovi slabi skalabilnosti in nezmožnosti prilagajanju ogromni količini zahtevkov, ki smo ji priča danes. Težava obstoječih arhitektur se kaže tudi v nezmožnosti definicije in razvoja vmesnikov API, ki se odzivajo glede na lokacijo. V magistrski nalogi smo definirali arhitekturo, ki omogoča definicijo in razvoj lokacijsko odvisnih vmesnikov API tipa REST. Ključna lastnost vmesnikov API je določanje pravic dostopa in prilagajanje delovanja glede na lokacijo uporabnika. Pri definiciji arhitekture smo sledili konceptom mikrostoritev s ciljem doseganja elastične skalabilnosti, visoke razpoložljivosti, razumljivosti in neodvisnosti. Z uporabo koncepta mikrostoritev ter ostalih arhitekturnih in načrtovalskih vzorcev smo zasnovali arhitekturo, ki temelji na konceptih oblaka in omogoča visoko horizontalno skalabilnost. Uspešnost arhitekture smo potrdili z verificiranjem zmožnosti definicije vmesnika API tipa REST z različnim obnašanjem glede na lokacijo uporabnika, z možnostjo samodejnega skaliranja mikrostoritev v oblaku in z verifikacijo načrtovalskih vzorcev.

Ključne besede

skalabilnost, arhitektura, mikrostoritve, lokacija, oblak, vmesnik API, REST

Abstract

We use services which use our location information to improve user experience on daily basis. Location-based services are well defined and can be built based on several existing architectures, especially in the form of a monolithic solution. Poor scalability and inability to handle immense loads are problems which existing architectures face today. Existing architectures also don't provide an option to define and develop location based APIs. In this master thesis we defined an architecture which allows the definition and development of APIs that respond based on user location. Key feature of the API is access control and adjusting functions based on user location. We based the architecture on microservices with the aim of achieving elastic scalability, high availability, understandability and decoupling. By using microservices and other architectural design patterns we designed an architecture which uses cloud concepts and enables high horizontal scalability. We validated the success of the architecture design by verifying the possibility of implementing an REST API with different behavior depending on the user location, the possibility of automatic scaling of microservices in the cloud and by verifying the design patterns.

Keywords

scalability, architecture, microservices, location, cloud, API, REST

Poglavje 1

Uvod

1.1 Motivacija

V času mobilnih aplikacij in spletnih storitev med bolj popularne štejemo lokacijske storitve (angl. Location-based services), ki uporabljajo informacijo o lokaciji za izboljšanje uporabniške izkušnje in nadzor nad funkcionalnostmi sistema. Lokacijske storitve informacijo o lokaciji navadno pridobijo na podlagi informacij iz omrežja, triangulacije mobilnega signala ali pa s pomočjo vgrajenih sprejemnikov GPS v mobilnih napravah. Lokacijske storitve so dobro opredeljene [1, 2, 3] in zanje obstaja več monolitnih arhitektur [4, 5, 6, 7], ki pa niso skalabilne in ne omogočajo definicije ter razvoja vmesnikov API tipa REST. Slaba skalabilnost monolitnih arhitektur se odraža pri nezmožnosti skaliranja aplikacij oz. slabi izrabi virov pri horizontalnem skaliranju. Definicija in razvoj vmesnikov API sta ključna pri sledenju najnovejšim arhitekturnim in implementacijskim vzorcem.

Veliko število uporabnikov in ogromne obremenitve strežnikov sta prinesla nove zahteve in razvoj naprednejših arhitektur. Podjetja, kot so Google[1], Twitter[2], eBay[3] in Amazon[4], so napredovali od velikih monolitnih arhitektur k arhitekturi mikrostoritev (angl. Microservices) [8, 9], katere mikrostoritve so implementirane z različnimi programskimi jeziki. Mikrostoritve omogočajo visoko skalabilnost, minimalno porabo virov, razpoložljivost in zanesljivost delovanja v sodobnih oblačnih infrastrukturah IaaS in PaaS. Mikrostoritve so navadno izpostavljene preko vmesnikov API tipa REST, preko katerih se tudi medsebojno povezujejo. Premišljena definicija in razvoj vmesnikov API je ključna za razvoj

[1] `http://www.google.com`
[2] `http://www.twitter.com`
[3] `http://www.ebay.com`
[4] `http://www.amazon.com`

posameznih mikrostoritev in ohranjanje njihove uporabnosti v prihodnjih zahtevah ter spremembah aplikacije. Nepremišljena definicija in razvoj vmesnikov API pa lahko prinese obilico nevšečnosti, potrebo po popravkih ali ponovno zasnovo ter implementacijo mikrostoritve.

1.2 Sorodna dela

Obstoječe monolitne arhitekture lokacijskih storitev [4, 6, 10] so nepopolne in ne izpostavljajo lokacijsko odvisnih vmesnikov API tipa REST. Mikrostoritve, ki pa navadno izpostavljajo vmesnike API tipa REST, so trenutno zelo aktualna tema, kar je razvidno iz obilice člankov in del s tega področja [9, 11, 12, 13, 14, 15]. V nadaljevanju je podan pregled ključnih sorodnih del. V prvem delu sta opisani dve že zasnovani arhitekturi lokacijskih storitev, na koncu pa so podana še aktualna dela s področja razvoja vmesnikov API in mikrostoritev, ki so pomembna za zasnovo naše arhitekture.

V članku [6] je predstavljena preprosta arhitektura lokacijske storitve za anonimno komuniciranje med dvema entitetama (objavitelj in naročnik). Avtorji so arhitekturo implementirali z uporabo platforme Java ME in podali vmesnik storitve. Sama arhitektura kljub svoji preprostosti daje idejo o osnovnih zahtevah ter funkcionalnostih lokacijske storitve in prikazuje preprost način implementacije s platformo Java. Za razliko od našega pristopa, rešitev v tem članku ne omogoča definicije lokacijsko odvisnih vmesnikov API, obenem tudi ni prilagojena izvajanju v oblaku, saj ne omogoča učinkovitega skaliranja ter minimalne porabe virov.

V nasprotju z arhitekturo, opisano v prejšnjem odstavku, arhitektura v članku [4] daje obsežnejšo rešitev za lokacijske storitve v obliki objavljanja in naročanja (angl. Publish/Subscribe). Avtorji opredelijo prostorske atribute dogodkov, ki so potrebni za delovanje in določanje ustreznosti lokacije uporabnika. Opisana arhitektura daje širšo sliko o problemu, vendar predstavlja monolitno rešitev in ne sledi principu mikrostoritev, kar posledično pomeni slabo skalabilnost arhitekture ter neprilagojenost izvajanju v oblaku. Poleg tega arhitektura ne omogoča razvoja lokacijsko odvisnih vmesnikov API.

Članek [10] opisuje lokacijski vmesnik API skupaj z implementacijo primera, ki temelji na platformi Java ME. Sama ideja avtorjev je predstavitev vmesnika, ki omogoča uporabo zemljevidov, geokodiranja in navigacije, za uporabo v naprednejših lokacijskih storitvah, kar bi močno zmanjšalo čas razvoja mobilnih aplikacij. Vmesnik omogoča standardiziran dostop do lokacije mobilne naprave v realnem času, spremljanje prihoda v območje in shrambo točk interesa. Sam vmesnik je odličen za implementacijo odjemalcev lokacijskih storitev na mobilnih napravah in nam daje idejo o zmožnostih odjemalcev na mobilnih napravah.

V knjigi [16] so predstavljeni vmesniki API tipa REST in množica pravil za razvoj le-teh, ki so bila pripravljena na podlagi najboljših in uveljavljenih praks. Avtor skupaj z zasnovo naslovov URI in uporabo protokola HTTP predstavi še načine za podajanje sporočil ter metapodatkov v glavah zahtevkov. V delu so poudarjene težave pomanjkanja standardov na tem področju in predlagana pravila, ki naslavljajo te težave. Avtor predstavi pomembnost razvoja in razmišljanja o vmesnikih API namesto implementacije, ki navadno ni težavna.

V obširnem delu [9] so opisane mikrostoritve, njihova integracija in skaliranje. Mikrostoritve so manjše avtonomne storitve, ki se medsebojno povezujejo. Avtor opiše prednosti, zaradi katerih so postale izjemno priljubljene med razvijalci in arhitekti. Predstavljena je integracija s poudarkom na vmesnikih API tipa REST, ki so trenutno eni izmed bolj priljubljenih. Skaliranje in zagotavljanje razpoložljivosti skupaj z ukrepi sta predstavljeni na koncu dela in dajejo dobre napotke za snovanje arhitekture.

Mikrostoritve v smislu samodejnega skaliranja in prilagajanja izvajanju v računalniškem oblaku so opisane v delu [13]. Ohlapna povezava mikrostoritev omogoča dinamičnejše objavljanje v oblaku in skaliranje posameznih mikrostoritev glede na obremenitve v nekem trenutku. Doseganje visoke razpoložljivosti in elastična skalabilnost sta zelo pomembni lastnosti naše arhitekture, ki sta bili premišljeno doseženi pri zasnovi le-te. V članku je prikazano tudi samo spreminjanje števila instanc mikrostoritev v oblaku pri samodejnem skaliranju glede na obremenitve, ki je eden izmed možnih načinov prikaza delovanja arhitekture.

Serija sedmih člankov [15], ki opisujejo vse od ideje mikrostoritev do izbire strategije za objavljanje le-teh, je povzetek in celosten primer zasnove arhitekture mikrostoritev, ki je služila kot opora pri zasnovi lastne arhitekture. Avtor je opisal težave monolitnih arhitektur in razmišljanje arhitektov pri prehodu na novo arhitekturo mikrostoritev. Članek sicer povzema ugotovitve in ideje mnogih drugih avtorjev, vendar lahko služi kot odlično pregledno delo.

Iz zgornjega pregleda sorodnih del je razvidno, da visoko skalabilna arhitektura za razvoj lokacijsko odvisnih vmesnikov API, kot je predlagana v tej magistrski nalogi, še ne obstaja.

1.3 Cilji in prispevki

V magistrskem delu smo zasnovali visoko skalabilno arhitekturo za razvoj lokacijsko odvisnih vmesnikov API, ki temelji na najsodobnejših razvojnih konceptih, omogoča elastično skalabilnost in minimalno porabo virov. Ključna lastnost lokacijsko odvisnih vmesnikov API je določanje pravic dostopa in prilagajanje delovanje glede na lokacijo uporabnika.

V prvi fazi magistrske naloge smo analizirali aktualne arhitekture drugih storitev, ki sledijo vzorcu mikrostoritev in najnovejših praks. Na podlagi poglobljene analize drugih arhitektur smo zasnovali lastno visoko skalabilno arhitekturo za razvoj lokacijsko odvisnih vmesnikov API. V drugi fazi smo zasnovali način za podajanje lokacije vmesnika API, ki mora biti prilagodljiv na večje število lokacij. Na koncu smo zasnovano arhitekturo implementirali z uporabo platforme Java EE in jo konfigurirali z ogrodjem KumuluzEE [11, 17]. Implementirano arhitekturo smo objavili v oblačni infrastrukturi PaaS. Uspešnost arhitekture smo v nadaljevanju potrdili z verificiranjem zmožnosti definicije vmesnika API tipa REST z možnostjo razlikovanja uporabnikov glede na njihovo lokacijo. Skalabilnost arhitekture pa smo preverili z možnostjo samodejnega skaliranja mikrostoritev v oblaku ob večjih obremenitvah.

Glavni prispevek magistrske naloge je zasnova visoko skalabilne arhitekture za definicijo in razvoj lokacijsko odvisnih vmesnikov API tipa REST, ki predstavlja napredek v primerjavi z dosedanjimi monolitnimi rešitvami. Rešitev je zasnovana v smeri minimalne porabe virov in omogoča elastično skalabilnost. Arhitektura temelji na najsodobnejših konceptih in vzorcih, razvita je z uporabo mikrostoritev. Drug večji prispevek magistrske naloge je implementacija arhitekture z najnovejšim ogrodjem KumuluzEE [11, 17]. Sama implementacija je dokumentirana in podrobno predstavljena v magistrskem delu.

1.4 Struktura magistrskega dela

Preostanek magistrskega dela je razdeljen na šest poglavij. Lokacijsko odvisni vmesniki API, lokacijske storitve in implementacija lastne lokacijsko odvisne anotacije so opisani v poglavju 2. Teoretična podlaga mikrostoritev, ki zajema opis arhitekture, primerjavo z drugimi arhitekturami, integracijo in skaliranje, je predstavljena v poglavju 3. Zasnova visoko skalabilne arhitekture za razvoj lokacijsko odvisnih vmesnikov API je opisana v poglavju 4. Implementacija arhitekture s platformo Java EE, konfiguracija z ogrodjem KumuluzEE in objava v oblačni infrastrukturi je podrobno dokumentirana v poglavju 5. V poglavju 6 je predstavljena verifikacija arhitekture v oblačni infrastrukturi PaaS. Na koncu v poglavju 7 pa so povzete ugotovitve magistrskega dela in predstavljene možnosti za nadaljnje delo.

Poglavje 2

Lokacijsko odvisni vmesniki

Aplikacijski programski vmesniki (angl. Application Programming Interface, API) izpostavljajo funkcionalnosti storitev na uveljavljene načine, ki omogočajo preprosto komunikacijo preko zahtevkov in odgovorov protokola HTTP. V tradicionalnih monolitnih arhitekturah se za komunikacijo s spletnimi storitvami uporablja protokol SOAP skupaj s sporočili tipa XML. V zadnjem času pa se za komunikacijo med odjemalci in storitvami večinoma uporablja arhitekturni način REST skupaj s sporočili tipa JSON. Za povezovanje in uporabo vmesnikov API je ključnega pomena dokumentacija, saj le-ta podaja informacije, ki so potrebne za uspešno integracijo. Dokumentacijo za vmesnike API tipa REST navadno podajamo s specifikacijo Swagger[1].

Lokacijsko odvisni vmesniki API omogočajo še več naprednih funkcionalnosti, ki so potrebne za izpostavitev lokacijskih storitev. Lokacijske storitve (angl. Location-based services) za izboljšanje uporabniške izkušnje in nadzor nad funkcionalnostmi sistema uporabljajo informacijo o uporabniški lokaciji. Integracija informacije o uporabniški lokaciji v delovanje storitev je uveljavljen koncept, kar je razvidno iz obilice lokacijskih storitev na področju navigacije, športa, zabave, družabnih omrežij, oglaševanja itd. Za nas najbolj zanimive so lokacijske storitve, ki lahko določajo pravice dostopa in prilagajajo delovanje na podlagi lokacije odjemalca. To poglavje v nadaljevanju podrobno opisuje lokacijske storitve in navaja naprednejše primere uporabe. V poglavju je predstavljen način podajanja lokacijske odvisnosti in prilagajanja delovanja vmesnikov API tipa REST s specifikacijo Swagger, ki smo jo razširili z možnostjo dodajanja območij delovanja posameznih metod tipa REST. Na podlagi razširitve specifikacije je mogoče definirati lokacijsko odvisne vmesnike API tipa REST. Na koncu poglavja pa je predstavljena implementacija anotacije za dodajanje lokacijske odvisnosti na vmesnike API na podlagi platforme Java EE.

[1] `http://www.swagger.io/`

2.1 Opis lokacijskih storitev

Lokacijske storitve so definirane kot storitve, ki integrirajo lokacijo odjemalca in druge informacije za izboljšanje uporabniške izkušnje in dodajanje vrednosti sami storitvi [1, 2, 3]. Osrednja informacija lokacijskih storitev je torej lokacija odjemalca oz. uporabnika. Lokacija odjemalca se navadno pridobi s pomočjo vgrajenih sprejemnikov GPS v mobilnih napravah ali na podlagi informacij iz omrežja ob dostopu preko brskalnikov. V odvisnosti od arhitekture same storitve se lokacija bodisi pošlje na strežnik bodisi uporabi v odjemalčevi aplikaciji. Glede na število mobilnih naprav in njihovo zmožnostjo lokalizacije preko sprejemnikov GPS ni čudno, da so lokacijske storitve izredno popularne, kar je razvidno iz obilice namenskih lokacijskih aplikacij (Google Maps[2], Uber[3], Pokemon Go[4]).

Storitve izpostavljamo preko vmesnikov, trenutno najbolj popularni so vmesniki API tipa REST [9, 16, 18], zato smo se v magistrskem delu osredotočili na lokacijsko odvisnost le-teh. Za izpostavljanje naprednejših funkcij lokacijskih storitev morajo vmesniki API omogočati definicijo več primerov uporabe, ki omogočajo lažjo implementacijo lokacijske odvisnosti. V nadaljevanju so našteti in opisani posamezni primeri uporabe.

Zmožnost definicije lokacijske odvisnosti vmesnika API in izpostavljenih metod je ena izmed glavnih zahtev za razvoj lokacijskih storitev. Definicija območja delovanja oz. aktivnosti lokacijske storitve in primerjava le-tega z lokacijo odjemalca storitve je eden izmed glavnih primerov uporabe, na katerem sloni celotno delovanje storitve. Posamezne metode vmesnika API so lahko definirane za različna območja ne glede na območje delovanja vmesnika.

Zmožnost definicije načina določanja pravic dostopa glede na lokacijo odjemalca storitve je naslednji primer uporabe vmesnikov. Vmesnik API je lahko definiran na več območjih in glede na ta območja je treba razlikovati med posameznimi odjemalci in njihovimi težnjami po dostopu do vmesnika in izpostavljenimi metodami. Poleg onemogočanja dostopa do vmesnika in izpostavljenih metod pa je možno tudi prilagajanje delovanja storitve.

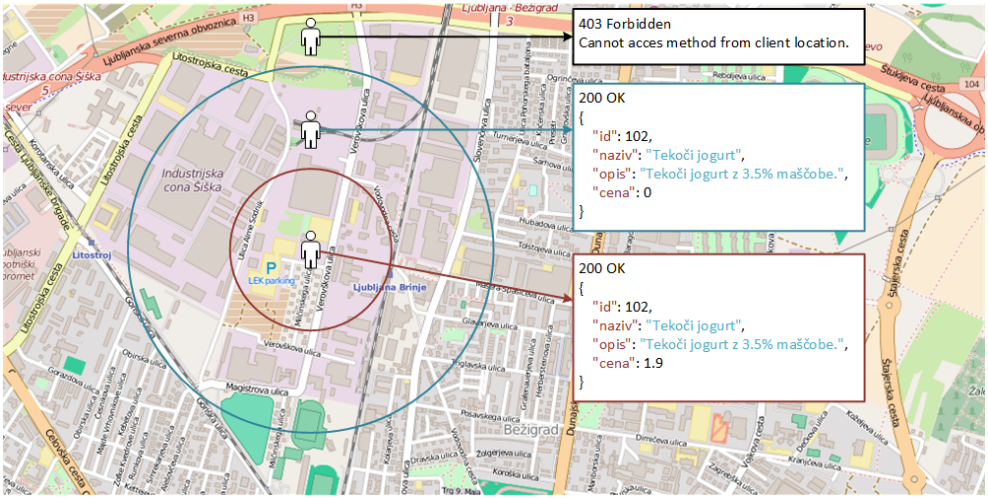
Naslednji primer uporabe je zmožnost definicije delovanja vmesnika in odgovorov metod glede na lokacijo in pravice dostopa odjemalca. Glede na izračunano območje nahajanja na podlagi lokacije odjemalca storitve je mogoče definirati prilagojene odgovore metod. Odgovori metod se lahko razlikujejo vse od podrobnosti entitet in števila polj pa vse do števila izbranih entitet posredovanih v obliki odgovora, kar je na kratko predstavljeno na sliki 2.1.

Zgoraj našteti primeri uporabe so nujno potrebni pri definiciji lokacijske odvisnosti

[2] `https://www.google.com/maps`

[3] `https://www.uber.com/`

[4] `http://www.pokemongo.com/`



Slika 2.1: Prilagajanje odgovora metode vmesnika glede na območje.

vmesnikov, ki izpostavljajo lokacijske storitve. V naslednjem poglavju je podana razširitev specifikacije Swagger za celovito definicijo lokacijsko odvisnih vmesnikov API tipa REST, ki vključuje te primere uporabe.

2.2 Definicija lokacijsko odvisnih vmesnikov

Lokacijske storitve lahko izpostavimo preko vmesnikov API, ki jih je pred implementacijo storitve dobro čim bolje definirati. Definicija vmesnikov omogoča lažji in hitrejši razvoj, ki najbolje upošteva smernice in prakse razvoja vmesnikov API tipa REST [16, 19]. Dokumentacija vmesnikov pa opisuje in omogoča lažje povezovanje in uporabo vmesnika drugim razvijalcem. V magistrski nalogi smo vmesnike API tipa REST izmed množice specifikacij definirali in dokumentirali z aktualno specifikacijo Swagger, ki omogoča celostno predstavitev vmesnikov.

V prilogi A.1 je definiran vmesnik API tipa REST s specifikacijo Swagger v orodju Swagger Editor[5]. Orodje omogoča definicijo vmesnikov, dopolnjevanje sintakse in preverjanje pravilnosti. To orodje poleg definicije ter preverjanja pravilnosti omogoča samodejno ustvarjanje strežniških aplikacij, ki izpostavljajo vmesnik API, in odjemalcev v različnih programskih jezikih.

Iz definicije vmesnika so razvidne vse informacije, ki so potrebne za implementacijo in uporabo tega vmesnika. Na voljo so splošne informacije o izpostavljenem vmesniku. Poleg

[5] `http://editor.swagger.io/`

osnovnih informacij so izpostavljene tudi informacije o posameznih metodah. Za vsako metodo so podane splošne informacije, parametri metode in možni odgovori. Metode so izpostavljene v skupinah glede na pot. Na koncu navadno definiramo skupne entitete, ki jih uporabljamo kot parametre ali odgovore metod. Specifikacija Swagger omogoča celostno definicijo vmesnikov API tipa REST, vendar brez lokacijskih odvisnosti. Če bi želeli definirati metodo na vmesniku, ki bi ustrezala primeru s slike 2.1, bi to storili zgolj z naštevanjem odgovorov brez natančne informacije o lokacijski odvisnosti le-teh.

Specifikacijo Swagger smo zato razširili z možnostjo dodajanja območij delovanja vmesnika in lokacijske odvisnosti posameznim metodam ter odgovorom. Na podlagi dopolnjene specifikacije je tako možno definirati lokacijsko odvisne vmesnike API tipa REST, kot je razvidno v prilogi A.2.

Specifikacijo Swagger smo lokacijsko razširili podobno, kot je to narejeno za varnost vmesnika. Na definicijo vmesnika smo dodali polje *locationDefinitions* za podajanje območij delovanja vmesnika in izpostavljenih metod. Posamezna območja so poimenovana s ključno besedo, opisana z daljšim opisom in imajo svoj tip. Trenutno je podprt samo tip krog, ki je definiran z zemljepisno širino, dolžino in polmerom v metrih. Tako je možno preprosto podati območje na uveljavljen način za podajanje lokacije oz. območja. V definiciji vmesnika se je v nadaljevanju možno poljubno sklicevati na ta območja delovanja.

Območja delovanja vmesnika smo nato uporabili za določitev območja delovanja posameznih metod tipa REST, ki jih ta vmesnik izpostavlja. Na vsaki metodi na vmesniku je možno definirati polje *onLocation* in v njem našteti vsa območja delovanja metode. Z dodatnima poljema smo zagotovili lokacijsko odvisnost in določili pravila dostopa glede na lokacijo odjemalca storitve do vmesnika ter metod tipa REST.

Lokacijske storitve lahko tudi prilagajajo delovanje glede na lokacijo uporabnika, kar je navzven razvidno predvsem iz odgovorov metod. Za ta primer uporabe smo dodali polje *onLocation* na posamezne odgovore metod. V tem polju je možno za posamezni odgovor našteti območja delovanja metode. Polje torej definira, v katerih območjih metoda vrača posamezni odgovor.

Z zgoraj naštetimi dodatnimi polji v specifikaciji Swagger smo naslovili primere uporabe lokacijskih storitev, ki so opisana v poglavju 2.1. Razširjena specifikacija omogoča podajanje lokacijske odvisnosti vmesnikov API in določa pravice dostopa ter prilagajanje delovanja vmesnikov API glede na lokacijo uporabnika. Z razširjeno specifikacijo Swagger je možno natančno definirati vmesnik, ki ustreza primeru s slike 2.1.

2.3 Implementacija anotacije

Na podlagi opisa lokacijskih storitev in definicije vmesnikov API iz predhodnih poglavij je v tem poglavju predstavljen praktičen primer implementacije lokacijsko odvisnega vmesnika API tipa REST z uporabo platforme Java EE. V poglavju so predstavljeni preučeni možni načini podajanja lokacijske odvisnosti vmesnikov in razlog za izbor implementacije lokacijske odvisnosti s pomočjo lastne anotacije. V nadaljevanju je nato dokumentirana implementacija lastne anotacije vse od zasnove projekta do dejanske logike v prestrezniku. Potem so predstavljeni načini definicije lokacijske odvisnosti ter območja delovanja na vmesnikih API tipa REST in njihovih izpostavljenih metodah. Na koncu poglavja pa je opisana predlagana razširitev konfiguracijske datoteke web.xml z možnostjo definicije območij delovanja aplikacije, ki izpostavlja vmesnike API.

Pred implementacijo logike za preverjanje ustreznosti lokacije odjemalca glede na definirana območja delovanja se poraja vprašanje o morebitnem preverjanju na odjemalcu. Preverjanje o ustreznosti lokacije bi lahko izvedli na uporabniškem vmesniku pred pošiljanjem klicev REST na storitve glede na začetna pridobljena območja delovanja. Preverjanje ustreznosti lokacije na odjemalcu prinaša občutno zmanjšanje dela na strani strežnika, kar je ena izmed ključnih prednosti. Kljub tej prednosti pa se pojavljajo velike neskladnosti in težave s tem pristopom. Pri neskladnosti delovanja je mišljeno predvsem na izvajanje logike na različnih lokacijah, poslovna logika in preverjanje varnosti bi se izvajala na mikrostoritvah, medtem ko bi se ustreznost lokacije preverjala na odjemalcu. Poleg neskladnosti delovanja ta pristop prinaša še dodatne metode na vmesnikih za pridobivanje območij delovanja ali dodatno mikrostoritev, ki skrbi za urejanje in pridobivanje območij delovanja. Preverjanje lokacije se iz teh razlogov zato preverja na strežniku, kar prinaša sicer več dela na že obremenjenih mikrostoritvah, vendar pa zmanjšuje število klicev REST s strani odjemalcev. Informacija o lokaciji se zato iz odjemalca pošlje v glavi zahtevka REST na vmesnik API, kjer se ta informacija nadaljnje obdela.

Možnih načinov za preverjanje ustreznosti lokacije odjemalca glede na območja delovanja je več. Na mikrostoritvah, ki izpostavljajo vmesnike API, lahko programsko preverimo lokacije. Ta način je preprost, vendar je potrebno dopolniti ali spremeniti vse metode na vmesnikih API, ki jim želimo dodati lokacijsko odvisnost. Malce boljša rešitev bi bila zasnove centralizirane lokacijske mikrostoritve, ki bi skrbela za izračun ustreznosti lokacije odjemalca. Ta rešitev bi prinesla le en dodaten klic lokacijske mikrostoritve iz posameznih metod, ki bi na podlagi lastne podatkovne zbirke območij delovanja preprosto izračunala ustreznost lokacije. Tak pristop prinaša dodatno potrebo po razvoju, urejanju in objavi še ene visoko skalabilne storitve. Za najbolj primerno rešitev se je izkazala implementacija lastne anotacije s pomočjo prestreznika za anotiranje vmesnikov in izpostavljenih metod. Ta način omogoča dodajanje lokacijske odvisnosti brez spremembe kode v posameznih me-

todah ter prestavitev ključne kode za preverjanje lokacije v lasten modul, ki ga je mogoče vključiti tudi v druge mikrostoritve.

Lastno anotacijo s pomočjo prestreznika smo zasnovali z možnostjo anotacije novih in že obstoječih razredov platforme Java EE brez potrebe po dodatni spremembi programske kode v metodah. Ta način omogoča uporabo anotacije @OnLocation na metodah in razredih platforme Java EE, ki izpostavljajo vmesnik API tipa REST, podobno kot lahko anotiramo varnostne omejitve dostopa z anotacijo @RolesAllowed. Anotacijo smo implementirali v lastnem modulu, katerega smo razvili v obliki modula Maven, ki omogoča vključitev v druge projekte in module. Na ta način je možno anotacijo za lokacijsko odvisnost uporabiti zgolj z dodajanjem odvisnosti na izdelan modul v konfiguracijski datoteki projekta Maven. Zasnova modula je privzeta, dodali pa smo odvisnost na artefakt kumuluzee-cdi-weld, ki omogoča uporabo tehnologije CDI 1.2 platforme Java EE za dodajanje kontekstov in vključevanje odvisnosti.

Anotacija za lokacijsko odvisnost vmesnikov API za svoje delovanje uporablja dve interni entiteti (entiteta Location in eniteta Area), ki sta prikazani v izseku 2.1. Entiteta Location predstavlja lokacijo, definirano z zemljepisno širino in dolžino. Ta entiteta se uporablja za zapis lokacije odjemalca in izračun razdalje med dvema lokacijama. Entiteta Area predstavlja območje, definirano z imenom, polmerom in zemljepisno dolžino ter širino. Ta entiteta se uporablja za zapis območij delovanja vmesnikov in izračun ustreznosti lokacije odjemalca glede na območje.

```
public class Location {
    private double latitude;
    private double longitude;
    ...
}

public class Area extends Location {
    private String name;
    private double radius;
    ...
}
```

Izsek 2.1: Podatkovni model anotacije za lokacijsko odvisnost.

Razdalja med dvema lokacijama, podanima z zemljepisno širino in dolžino, se izračuna s pomočjo formule haversine (angl. Haversine formula), ki v izračunu upošteva krivino krogle in predstavlja najkrajšo razdaljo po plašču krogle. Poenostavljena formula, ki smo

jo implementirali s platformo Java EE, je podana z

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{2.1}$$

kjer je r radij Zemlje v metrih, φ_1, φ_2 zemljepisna širina točk v radianih in λ_1, λ_2 zemljepisna dolžina točk v radianih. Rezultat d je podan v metrih, tako kot smo podali radij Zemlje.

V izseku 2.2 je prikazana definicija anotacije za lokacijsko odvisnost vmesnikov API v platformi Java EE. Anotacija je povezana in implementirana s prestreznikom. Z anotacijo je možno anotirati razrede vmesnikov in tudi posamezne metode na teh razredih. Sama anotacija se ohranja na razredih in metodah tudi tekom izvajanja, ker je informacija potrebna za pravilno izvajanje prestreznika. Anotacija ima atribute ime, zemljepisno širino, dolžino in polmer. S temi atributi je možno definirati območja z lokacijo in polmerom, ter jih tudi poimenovati. Nobeden od atributov anotacije za lokacijsko odvisnost ni obvezen, kar pa je podrobneje predstavljeno v nadaljevanju.

```
@InterceptorBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface OnLocation {
    @Nonbinding
    String name();

    @Nonbinding
    double latitude();

    @Nonbinding
    double longitude();

    @Nonbinding
    double radius();
}
```

Izsek 2.2: Definicija anotacije za lokacijsko odvisnost.

Prestreznik anotacije za lokacijsko odvisnost, ki je prikazan v dodatku B, prestreže klic razredov ali metod označenih kot lokacijsko odvisne in vrne odgovor z napako, ali pa posreduje zahtevek na klicano metodo. Prestreznik smo omogočili z navedbo v konfiguracijski datoteki beans.xml v mapi za meta informacije znotraj projekta. Prestreznik najprej

preveri, ali zahtevek REST vsebuje informacijo o zemljepisni dolžini ter širini odjemalca in v primeru napačnega formata ali odsotnosti te informacije vrne odgovor z opisno napako. Naslednji korak v izvajanju prestreznika je pridobivanje informacije o anotaciji za lokacijsko odvisnost iz razreda ali metode. Na podlagi informacije o območju delovanja anotacije se izvajanje razveja v tri veje. Pri izvajanju prve veje se preveri, ali je območje definirano kar z informacijo o zemljepisni širini, dolžini in polmerom. V primeru da lokacija pridobljena iz zahtevka REST ustreza definiranemu območju, se izvede posredovanje zahtevka na klicano metodo, drugače pa se vrne odgovor z opisno napako. Pri drugi veji izvajanja se preveri, ali je območje definirano z imenom. V tem primeru se lokacija iz zahtevka REST primerja z vsemi območji delovanja, navedenimi v konfiguracijski datoteki za območja, in v primeru ujemanja s pravilno poimenovanim območjem se zahtevek posreduje na klicano metodo. Pri zadnji veji izvajanja pa se izmed vseh območij v konfiguracijski datoteki za območja preveri ustreznost lokacije iz zahtevka REST, ter se v primeru zadetka območja izvajanje nadaljuje na klicani metodi. Zahtevku REST pa se v tem primeru doda tudi informacija o območju delovanja, ki mu ustreza lokacija odjemalca, za kasnejšo uporabo znotraj klicane metode. V primeru neskladja z območji delovanja ali odsotnosti konfiguracijske datoteke za območja se vrne odgovor z opisno napako.

Izsek 2.3 prikazuje način uporabe implementirane anotacije za lokacijsko odvisnost. Metode na razredu, ki implementira vmesnik, so anotirane na tri možne načine, vsakemu ustreza ena izmed vej opisanih v prejšnjem odstavku. Anotacija na metodi za pridobivanje informacije o obstoju izdelka je definirana kar z zemljepisno širino, dolžino in polmerom, ki skupaj označujejo območje delovanja. Anotacija na metodi za pridobivanje izdelka je definirana z imenom območja, ki se izvede le, če lokacija odjemalca ustreza območju s podanim imenom iz konfiguracijske datoteke za območja. Zadnja anotacija na metodi za pridobivanje vseh izdelkov pa je definirana brez parametrov, kar sproži preverjanje ustreznosti lokacije odjemalca glede na vsa območja v konfiguracijski datoteki.

V izseku 2.4 je prikazana vsebina konfiguracijske datoteke location-definitions.xml, ki predstavlja način podajanja lokacijske odvisnosti za uporabo v prestrezniku anotacije za lokacijsko odvisnost. Posamezna območja delovanja so definirana znotraj oznak area, ki omogočajo preslikavo v objekt Area iz podatkovnega modela anotacije. Trenutno je konfiguracijsko datoteko možno vključiti v mapo projekta WEB-INF, ki uporablja implementirano anotacijo. Za nadaljnjo uporabo pa predlagamo vključitev vsebine datoteke v opisno datoteko za objavo web.xml. Na podoben način, kot so definirane varnostne omejitve, bi lahko podali še lokacijsko odvisnost vmesnikov, s tem pa bi zmanjšali število potrebnih datotek in povezano kompleksnost z uporabo anotacije za lokacijsko odvisnost.

```
@RequestScoped
@Path("/izdelki")
public class IzdelekVir {
    @HEAD
    @Path("/{id}")
    @OnLocation(latitude = 46.076471, longitude = 14.499496,
        radius = 100)
    public Response obstaja(@PathParam("id") Integer id) {
        ...
    }

    @GET
    @Path("/{id}")
    @OnLocation(name = "obmocje2")
    public Response pridobi(@PathParam("id") Integer id) {
        ...
    }

    @GET
    @Path("")
    @OnLocation
    public Response pridobiVse() {
        ...
    }
}
```

Izsek 2.3: Anotacija metod z anotacijo za lokacijsko odvisnost.

S tem smo zaključili s podrobnim opisom možne implementacije lokacijsko odvisnih vmesnikov API tipa REST s platformo Java EE. Naša implementacija anotacije za lokacijsko odvisnost omogoča preprosto ponovno uporabo v drugih projektih in konfiguracijo s pomočjo ogrodja KumuluzEE. Kljub natančni zasnovi anotacije, pa bi želeli razširiti opisno datoteko web.xml, ki vsebuje ključne informacije, potrebne za objavo projekta na strežnikih in oblačnih strukturah, z možnostjo definicije območij delovanja vmesnikov.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<location-definitions>
    <area>
        <name>obmocje1</name>
        <latitude>46.076471</latitude>
        <longitude>14.499496</longitude>
        <radius>100</radius>
    </area>
    <area>
        <name>obmocje2</name>
        <latitude>46.076471</latitude>
        <longitude>14.499496</longitude>
        <radius>200</radius>
    </area>
</location-definitions>
```

Izsek 2.4: Definicija območij delovanja v konfiguracijski datoteki location-definitions.xml.

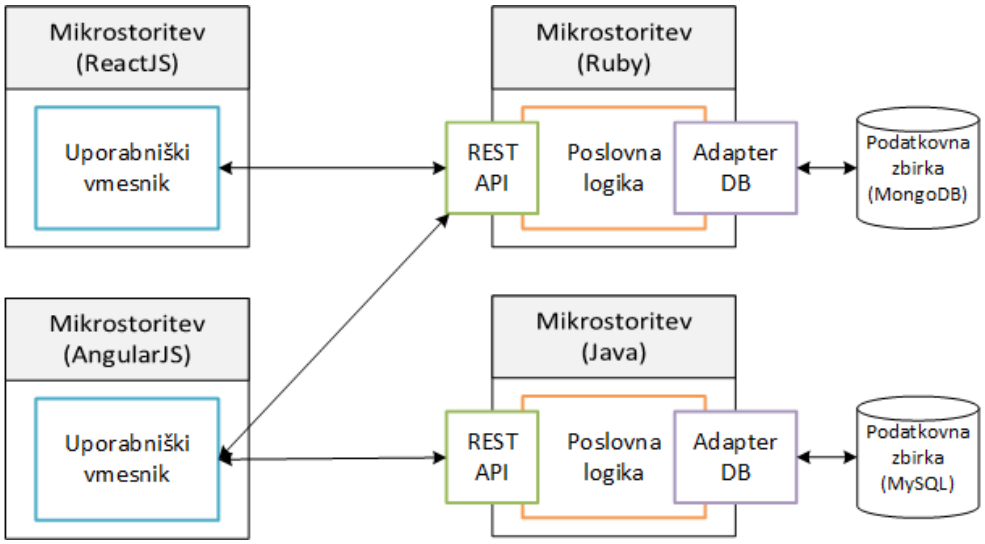
Poglavje 3

Koncepti mikrostoritev

Mikrostoritve (angl. Microservices) so majhne in neodvisne storitve, ki med sabo komunicirajo preko vmesnikov API. Posamezne mikrostoritve implementirajo manjša opravila in predstavljajo zaokrožene funkcionalne celote. Mikrostoritve so navadno zadolžene za pridobivanje, urejanje in dodajanje ene entitete podatkovnega modela. Arhitektura mikrostoritev omogoča visoko skalabilnost in je kot taka primerna za današnje oblačne infrastrukture PaaS. To poglavje v nadaljevanju podrobno opisuje mikrostoritve in primerjavo s tradicionalno monolitno arhitekturo. V poglavju je predstavljeno skaliranje mikrostoritev, kocka skaliranja, skaliranje mikrostoritev v oblačnih strukturah in arhitekturni elementi, ki vse to omogočajo. Na koncu poglavja pa je predstavljeno ogrodje KumuluzEE, ki avtomatizira proces objave in konfiguracije aplikacij, ki temeljijo na platformi Java EE.

3.1 Opis mikrostoritev

Projekti z implementacijo in dodajanjem novih funkcionalnosti sčasoma postanejo ogromne monolitne strukture. Projekti postanejo preveč kompleksni za preprosto spreminjanje in razumevanje, kljub prizadevanjem po čisti kodi in organizirani strukturi. Kompleksnost projektov delno naslovimo z delitvijo na module in podobna ideja stoji tudi za mikrostoritvami. Mikrostoritve, ki med sabo komunicirajo preko vmesnikov, naslavljajo težave velikih monolitnih struktur z razčlenitvijo na funkcionalne celote. Arhitektura mikrostoritev omogoča lažje iskanje po funkcionalnostih, velikost mikrostoritev pa omogoča lažje spreminjanje in dodajanje novih funkcionalnosti. Pojavlja se vprašanje o obsegu posameznih mikrostoritev [9]. Mikrostoritve naj bi obsegale pridobivanje, urejanje in dodajanje ene entitete podatkovnega modela oz. manjšo celostno funkcionalnost. Dobra razdelitev na mikrostoritve je lahko težavna in je v veliki meri odvisna predvsem od podatkovnega



Slika 3.1: Implementacija arhitekture z različnimi tehnologijami.

modela.

Mikrostoritve izpostavljajo uporabniške vmesnike ali vmesnike API. Preko slednjih mikrostoritve medsebojno komunicirajo in se povezujejo. Stroga ločitev mikrostoritev omogoča implementacijo z različnimi tehnologijami, neodvisno spreminjanje in ločeno objavo. Na tem mestu je potrebno omeniti še ločitev podatkovnih zbirk mikrostoritev. Skupna podatkovna zbirka bi za mikrostoritve predstavljala ozko grlo, zato mikrostoritve tipično uporabljajo ločene podatkovne zbirke. Posamezne entitete podatkovnega modela se zato nahajajo v lastnih podatkovnih zbirkah. Uporaba različnih tehnologij za posamezne mikrostoritve in podatkovne zbirke prikazuje slika 3.1.

Arhitektura mikrostoritev prinaša mnogo prednosti ne le pri zmogljivosti, temveč tudi pri zasnovi, razvoju in objavi mikrostoritev [9, 13, 14, 15, 20]. Glavna prednost je naslavljanje kompleksnosti strukture. Večje neobvladljive storitve so razbite v več manjših storitev, ki z medsebojno komunikacijo nudijo iste funkcionalnosti. Velikost mikrostoritev omogoča lažje razumevanje funkcionalnosti in hitrejši razvoj. Poleg lažjega razvoja je mikrostoritve možno razvijati neodvisno med sabo z ločenimi različicami. Neodvisnost prinaša tudi razvoj v različnih tehnologijah, kar omogoča sledenje najnovejšim trendom in uporabo najbolj primernih tehnologij bodisi glede na zmogljivost bodisi glede na znanje razvijalcev. Ob spreminjanju obstoječe mikrostoritve je treba v oblak ali na strežnik objaviti le spremenjeno mikrostoritev in ne celotne storitve kot pri monolitni arhitekturi. Ločitev storitev prinaša tudi odpornost na napake, saj napaka ene storitve ne zruši celotnega sistema, ampak ostale mikrostoritve delujejo naprej nemoteno. Še ena pomembna

prednost je možnost skaliranja mikrostoritev v današnjih oblačnih strukturah PaaS. Visoka skalabilnost mikrostoritev omogoča visoko razpoložljivost in zmogljivost celotnega sistema, kar je podrobneje opisano v poglavju 3.5.

Mikrostoritve prinašajo prednosti, vendar hkrati dodajajo nove težave in imajo svoje pomanjkljivosti. Porazdeljen sistem in dodatna komunikacija med mikrostoritvami vnaša režijsko delo zaradi dodatnih struktur v arhitekturi in potrebo po implementaciji klicev na druge mikrostoritve. Poleg tega je velikost v pomnilniku in potreba po virih večja pri uporabi te arhitekture. To dodatno delo in večjo porabo virov je treba vzeti v račun pri iskanju primerne arhitekture. Spreminjanje neodvisnih funkcij je navadno preprosto, vendar pa lahko postane zelo zapleteno pri odvisnih funkcijah in modelih, predvsem pri poizkusu objave novih različic mikrostoritev v oblaku ali na strežniku. Pomanjkljivost mikrostoritev se pojavlja tudi pri ločenih podatkovnih zbirkah, saj vnašajo potrebo po implementaciji vodenja transakcij, kar je lahko zamudno in zelo težavno za testiranje vseh možnih vej izvajanja. Objava posameznih mikrostoritev v oblaku je preprosta, vendar pa se pojavlja dodatno delo pri vzpostavitvi dodatnih arhitekturnih elementov, ki so podrobneje predstavljeni v poglavju 3.6. Do dodatnega dela in potrebe po znanju pa ne prihaja le pri objavi, temveč tudi pri spremljanju in vodenju teh struktur.

Implementacija kompleksnih aplikacij oz. storitev je navadno težavna, vendar pa lahko nekatere težave naslovimo z arhitekturo mikrostoritev. Prednosti arhitekture mikrostoritev postanejo razvidne in odtehtajo slabosti šele v kompleksnejših sistemih, kjer bi drugače imeli ogromno monolitno arhitekturo. Tradicionalne monolitne arhitekture imajo smisel v manjših storitvah z manj predvidenega prometa in uporabnikov, kot je to podrobneje opisano v poglavju 3.2.

3.2 Primerjava z monolitno arhitekturo

Monolitna arhitektura je ena najbolj uporabljenih arhitektur za razvoj storitev oz. aplikacij. Pri tradicionalni monolitni arhitekturi celotno aplikacijo implementiramo znotraj enega projekta in jo združeno, arhivirano v celoto objavimo na strežnik. V platformi Java EE aplikacijo implementiramo v več modulih znotraj projekta in vse skupaj združimo v skupni arhiv EAR. Monolitna aplikacija ima veliko prednosti, vendar se lahko sprevrže v zelo obsežen projekt, kar pa prinaša obilico težav predvsem v fazi razvoja, testiranja in objave v oblačno strukturo ali na strežnik.

Osrčje monolitne aplikacije predstavlja poslovna logika implementirana v več modulih, ki definirajo storitve, poslovne tipe in dogodke [15, 21, 22]. Poslovno logiko lahko izpostavimo s pomočjo različnih vmesnikov API ali pa uporabimo preko uporabniškega vmesnika, ki ga implementiramo znotraj svojega modula. Poslovna logika z zunanjim svetom komu-

nicira tudi preko drugih vmesnikov, kot so vmesniki za podatkovne zbirke, komponente za izmenjavo sporočil in odjemalci za integracijo z drugimi storitvami. Število vmesnikov v monolitni aplikaciji nam daje dobro idejo o svoji obsežnosti in zapletenosti.

Glavna prednost monolitnih arhitektur je v njihovem preprostem celostnem razvoju, testiranju, objavljanju in skaliranju [15, 21]. Razvoj monolitnih aplikacij je preprost in izvedljiv v današnjih razvojnih okoljih, saj vso funkcionalnost in spremembe dodajamo v en projekt, ki ga imamo v delovnem okolju. Ne le, da je razvoj monolitnih aplikacij izvedljiv v današnjih razvojnih okoljih, temveč so današnja okolja prilagojena razvoju monolitnih aplikacij. Testiranje monolitne aplikacije je prav tako preprosto, ker celotna aplikacija teče na enem strežniku in med posameznimi moduli ni dodatne komunikacije preko omrežja. Razvojna okolja navadno tudi omogočajo orodja za objavo aplikacije na strežnikih, tako na razvojnih (lokalnih) kot tudi na oddaljenih. Objava je preprosta tudi iz razloga, ker imamo celotno aplikacijo v obliki paketa in lahko ta paket tudi sami prenesemo v strežniško okolje ali v oblačno strukturo PaaS. Monolitne aplikacije lahko skaliramo vertikalno z dodajanjem virov ali pa horizontalno z izvajanjem več instanc za razporejevalnikom bremena, kot je podrobneje opisano v poglavju 3.5.

Prednosti monolitne aplikacije hitro izzvenijo ob povečanju velikosti aplikacije. Z razvojem aplikacije ta postaja vse bolj zapletena, težje razumljiva in težavna za razvoj, kar je razvidno iz slabše preglednosti in težjega vključevanja novih razvijalcev na projekt. Z večanjem obsega aplikacije se poveča tudi kompleksnost objave na strežnik in hitrost razvoja v razvojnem okolju. Testiranje prav tako postane težavno predvsem zaradi nezmožnosti stalne in hitre objave na strežnik. Objava na strežnik postane dolgotrajen proces, kar povzroči slabšo produktivnost razvijalcev. Skaliranje monolitne aplikacije je sicer preprosto, vendar samo v eni dimenziji. Skaliranje po osi x, ki je podrobneje opisano v poglavju 3.5, z ustvarjanjem več instanc aplikacije omogoča večjo skalabilnost, vendar na račun zelo velike porabe infrastrukturnih virov. Aplikacija sčasoma postane ogromna, poveča pa se tudi poraba virov v infrastrukturi. Še ena slabost monolitne arhitekture pa je vezava na tehnologijo oz. programski jezik, ki ga med samim razvojem ni možno zamenjati brez ponovne implementacije celotne aplikacije.

Glede na opis v tem poglavju so razvidne ključne razlike med monolitno arhitekturo in arhitekturo mikrostoritev. Razvidne so prednosti manjših monolitnih aplikacij, ki pa jih v neki točki obsega močno pretehtajo slabosti. Ob dosegu prevelikega obsega monolitne aplikacije je navadno čas za razmislek o razgradnji na mikrostoritve, še bolje pa je to storiti že na začetku razvoja v fazi zasnove, saj to skrajša razvojni čas. Ob razgradnji monolitne aplikacije se najhitreje opazijo prednosti, kot so hitrejša odzivnost razvojnega okolja, lažje razumevanje same mikrostoritve in krajši čas objave. Mikrostoritve pa ne omogočajo le skaliranje po osi x, kot to omogočajo monolitne arhitekture, temveč tudi skaliranje po osi y in z. Te tri osi skaliranja pa so ključnega pomena pri zagotavljanju

visoke skalabilnosti arhitekture, ki je danes nujna za doseganje visoke razpoložljivosti in zagotavljanje zmogljivosti sistema.

3.3 Primerjava z arhitekturo SOA

Storitveno usmerjena arhitektura (angl. Service-Oriented Architecture, SOA) je arhitekturni vzorec, pri katerem storitve medsebojno komunicirajo preko omrežja in skupaj zagotavljajo celostno množico funkcionalnosti velike storitve ali aplikacije [23, 24]. Storitve so ločene in zaključene celote, ki med sabo navadno komunicirajo preko protokola SOAP (angl. Simple Object Access Protocol) in so definirane z vmesnikom WSDL (angl. Web Service Definition Language), ki nudi natančne informacije o vratih, operacijah, tipih, zahtevah in odgovorih. Arhitektura SOA naslavlja težave monolitnih arhitektur z idejo o ponovni uporabi storitev, kar je tudi ideja samih mikrostoritev. Z uporabo izpostavljenih storitev se spodbujata granularnost in neodvisnost, kar prinaša podobne prednosti kot arhitektura mikrostoritev. Šibka sklopljenost je torej eden izmed ključnih principov sodobnih načinov razvoja in objave storitev. Poleg šibke sklopljenosti se pri obeh arhitekturah pojavljajo še drugi principi, kot so abstrakcija implementacije, storitve brez stanj in preprosto odkrivanje storitev.

Arhitektura SOA [23, 24] je prvotno namenjena razgradnji storitev znotraj organizacije in nudi podporo delovanju celotne organizacije. Z razgradnjo in ponovno uporabljivostjo storitev organizacije se pripomore k hitrejšemu poslovnemu odzivnemu času in zmanjšanju stroškov infrastrukture z manjšo porabo virov. Storitve v arhitekturi so implementirane v ločenih projektih z različnimi tehnologijami, izpostavljene preko vmesnikov različnih tipov in objavljene v lastni ali gostujoči infrastrukturi. Storitve najpogosteje izpostavljamo preko vmesnikov SOAP, lahko pa tudi preko drugih vmesnikov z vmesnimi mediacijskimi storitvami, ki skrbijo za preoblikovanje sporočil. Komunikacija med vmesniki storitev poteka preko komunikacijskega kanala ESB, ki skrbi za neodvisno komunikacijo in omogoča uporabo različnih tehnologij. Kanal ESB (angl. Enterprise Service Bus) je torej ključnega pomena pri komunikaciji storitev in njihovih odjemalcev. Za arhitekturo SOA je značilna tudi stroga definicija izpostavljenih vmesnikov po uveljavljenih specifikacijah, kar omogoča povezljivost in preprosto uporabo storitev. Iz glavnih točk arhitekture SOA so torej razvidne ključne podobnosti in tudi razlike z arhitekturo mikrostoritev.

Arhitektura mikrostoritev uporablja druge protokole za medsebojno komunikacijo storitev in izpušča elemente arhitekture SOA, kot sta recimo specifikacija spletnih storitev WS in komunikacija preko kanala ESB. Kljub razliki v protokolih in izpuščanju elementov arhitekture SOA veliko poznavalcev trdi, da arhitektura mikrostoritev ni nič novega [9, 15], temveč je le še en specifičen primer arhitekture SOA. Obe arhitekturi uporabljata iste ali

podobne principe, kar potrjujejo trditve strokovnjakov. Kljub tej ugotovitvi je razvidno, da arhitektura SOA naslavlja razgradnjo storitev organizacije, medtem ko arhitektura mikrostoritev naslavlja razgradnjo posameznih storitev na manjše zaključene celote. Ta fina granularnost mikrostoritev je bolje opredeljena kot razgradnja v arhitekturi SOA. Slednja ugotovitev pa je mogoče razlog za pridobivanje na popularnosti arhitekture mikrostoritev v zadnjih letih, čeprav se arhitektura SOA uporablja že več kot desetletje.

3.4 Integracija mikrostoritev

Uporaba najbolj primernih tehnologij in konceptov je ključnega pomena za uspešno integracijo mikrostoritev med razvojem, objavo in vzdrževanjem arhitekture. Na podlagi izbranih tehnologij in konceptov bo temeljila celotna implementacija mikrostoritev, ki pa je lahko ob napačni izbiri zelo otežena. Tehnologije integracije se nanašajo na način izpostavitve vmesnikov, način prenosa podatkov, tipi podatkovnih zbirk, način vodenja različic in še mnogi drugi. Hkrati je treba upoštevati koncepte, kot so preprostost uporabe, skrivanje podrobnosti implementacije in tehnološka neodvisnost mikrostoritev. Snovanje načina integracije je torej zelo obsežno predvsem zaradi obilice odločitev glede ključnih tehnologij in pristopov.

Trenutno najbolj popularni so vmesniki tipa REST [9, 16, 25]. Arhitekturni način REST je preprost za razumevanje, ima pa tudi svoje omejitve. Ključne besede, kot so HEAD, GET, POST, PUT in DELETE, so razumljive in imajo natančno definirano obnašanje na virih, ki implementirajo vmesnike API. Poleg vmesnikov tipa REST obstajajo še vmesniki tipa SOAP, ki se uporabljajo predvsem v poslovnih aplikacijah z arhitekturo SOA, in oddaljeni klici procedur RPC, ki pa so večinoma odvisni od izbranega programskega jezika in zato niso najbolj primerni za arhitekturo mikrostoritev. Glede na priporočila in izbiro večjih podjetij so vmesniki tipa REST trenutno najbolj primerna izbira.

Zahteve in odgovori vmesnikov API so lahko različnih formatov [9, 16], za vmesnike SOAP se uporablja format XML, medtem ko se za vmesnike REST najpogosteje uporablja format JSON in tudi drugi tekstovni formati. Prednosti formata JSON sta preprostost in krajši zapis ter posledično krajši čas prenosa preko omrežja [26], kot je razvidno tudi v izsekih 3.1 in 3.2. Poleg tega je ta format preprost za uporabo na uporabniških vmesnikih s programskim jezikom JavaScript. Format XML je bolj generičen način prenosa podatkov z obsežnejšimi shemami, večjim naborom tipov in podporo uporabe imenskih prostorov. Za prevlado formata JSON za izmenjavo podatkov na vmesnikih tipa REST ima zasluge predvsem njegova preprostost in uporaba v knjižnicah programskega jezika JavaScript, hkrati pa tudi hitrost prenosa in manjša poraba virov.

Ne glede na izbiro tipa vmesnika in formata prenosa podatkov je treba poudariti, da arhitektura mikrostoritev omogoča uporabo različnih programskih jezikov za implementacijo posameznih mikrostoritev. Na podlagi ločitve zasnove in implementacije je možno stalno izbirati najnovejše programske jezike in njihova ogrodja za implementacijo novih storitev ali ponovno implementacijo starih storitev. Na ta način lahko organizacije sledijo najnovejšim trendom v razvoju, kar je nujno za poslovno uspešnost in ohranjanje konkurenčne prednosti.

```
<?xml version="1.0" encoding="UTF-8"?>
<izdelek>
    <id>1</id>
    <naziv>Jagodni jogurt</naziv>
    <opis>Naravni jogurt z okusom jagode.</opis>
    <cena>1.11</cena>
</izdelek>
```

Izsek 3.1: Sporočilo tipa XML.

```
{
    "id": 1,
    "naziv": "Jagodni jogurt",
    "opis": "Naravni jogurt z okusom jagode.",
    "cena": 1.11
}
```

Izsek 3.2: Sporočilo tipa JSON.

Posamezne storitve lahko med sabo integriramo tudi preko podatkovne zbirke. Urejanje podatkov o entitetah v skupni podatkovni zbirki je neka vrsta integracije, saj vse storitve dobijo informacijo o spremembi ob naslednjem dostopu do te entitete. Integracija s pomočjo podatkovne zbirke je mogoča za instance iste mikrostoritve, saj navadno uporabljajo eno kopijo podatkovne zbirke. Podatkovne zbirke storitev pa so v arhitekturi mikrostoritev ločene med sabo in prinašajo več dela. Mikrostoritve morajo vse težnje po spremembi entitet nasloviti na pristojno mikrostoritev, ki omogoča urejanje specifične entitete. Na podlagi tega je integracija preko podatkovne zbirke dosežena preko vmesnikov API. Prav zaradi ločenih podatkovnih zbirk pa je po drugi strani možno uporabljati zbirke različnih tipov, ki so najbolj primerne za primer uporabe entitete. Skupno podatkovno zbirko v arhitekturi mikrostoritev nadomestimo z več bolje prilagojenih podatkovnih zbirk, ki pa jih je treba usklajevati z dodatnimi klici na vmesnike.

Upoštevanje ponovne uporabe kode po principu DRY (angl. Don't Repeat Yourself) prinaša težave pri arhitekturi mikrostoritev. Same mikrostoritve so šibko sklopljene

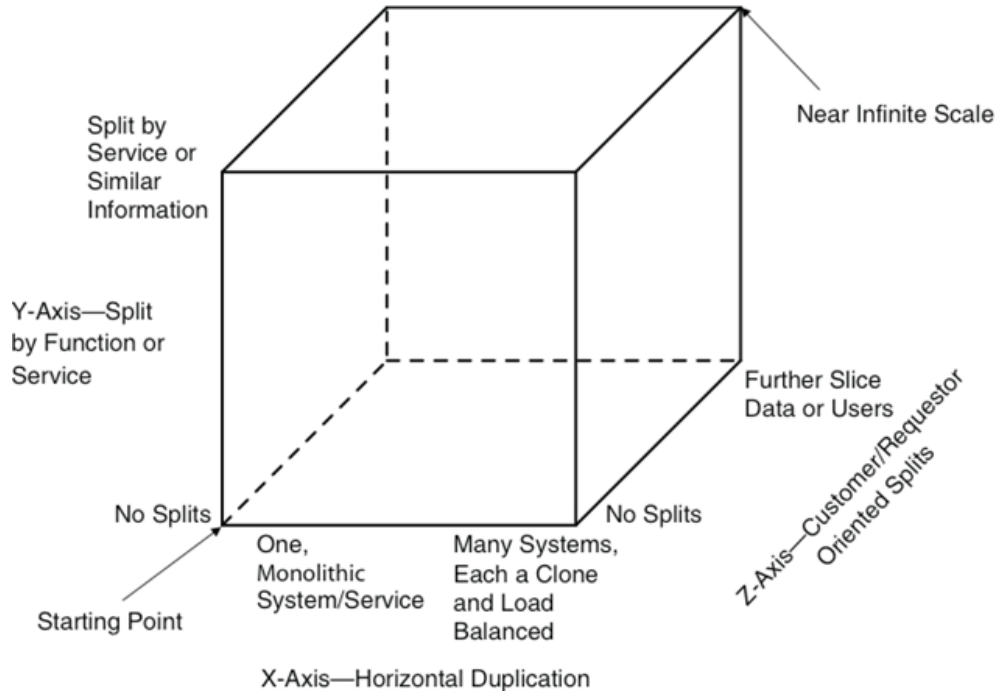
in uporaba skupne programske kode ima več posledic. Ena izmed vidnejših posledic je nezmožnost uporabe skupnih implementacij med mikrostoritvami implementiranimi z različnimi tehnologijami, razen v primeru izpostavljanja preko še ene mikrostoritve, kar pa lahko privede do še kompleksnejše arhitekture. Ob uporabi skupnih funkcionalnosti ali knjižnic v mikrostoritvah implementiranih z isto tehnologijo, pa je treba že ob najmanjši spremembi deljene kode ponovno objaviti vse te storitve. Upoštevanje principa DRY se zato priporoča znotraj mikrostoritev in ne deljeno med mikrostoritvami [9].

Z razvojem mikrostoritve lahko pride do potrebe po spremembi vmesnika API, kar prinaša potrebo po zasnovi načina vodenja različic. S spremembo vmesnika API so najverjetneje potrebne tudi spremembe na mikrostoritvah, ki se povezujejo na dani vmesnik. Hkratna posodobitev vseh storitev bi bila preobsežna, zato je v navadi hkratno izpostavljanje novih in starih vmesnikov, dokler vse mikrostoritve ne preidejo na novo različico vmesnika. V tem trenutku je potem mogoče vmesnik označiti kot zastarel in ga odstraniti. Označevanje in vodenje različic tekom razvoja mikrostoritev pa naj bi potekalo na uveljavljen ter konstanten način.

3.5 Skaliranje mikrostoritev

Skalabilnost je lastnost storitve ali sistema, ki omogoča rokovanje z vedno večjim številom uporabnikov, dela in obremenitve. Danes skaliramo storitve z namenom zagotavljanja visoke razpoložljivosti in zmogljivosti. Visoka razpoložljivost storitev je nujna za uporabnike, kljub napakam ki se pojavljajo v storitvi ali v infrastrukturi. Z večanjem števila uporabnikov in bremena je treba povečati zmogljivosti storitev in infrastrukture. Današnje oblačne infrastrukture IaaS in PaaS omogočajo veliko možnosti in orodij za skaliranje storitev. Najbolj preprost način skaliranja je povečevanje zmogljivost strojev v infrastrukturi oz. z vertikalnim skaliranjem. Vertikalno skaliranje lahko hitro postane zelo drago in neučinkovito, predvsem pri zagotavljanju visoke razpoložljivosti. Skalabilnost storitev je torej predvsem arhitekturni problem, ki ga je treba nasloviti že v fazi zasnove arhitekture storitve.

Umetnost skaliranja [27] je obsežnejše delo na temo skaliranja arhitektur, procesov in organizacij. Možni načini skaliranja vse od monolitne storitve do teoretičnega skoraj neskončnega obsega so prikazani na sliki 3.2, ki prikazuje kocko skaliranja. Skaliranje po osi x oz. horizontalno skaliranje je najpreprostejše izmed opisanih treh načinov skaliranj in omogoča izvedbo z ustvarjanjem več instanc storitve, ki so izpostavljene za preprostim razporejevalnikom bremena. Ideja pri skaliranju po osi y je v razgradnji storitve na manjše dele oz. mikrostoritve, ki naj zajemajo funkcionalne celote. Skaliranje po osi z pa predstavlja skaliranje storitev glede na podmnožice zahtevkov ali uporabnikov, ki se preko



Slika 3.2: Kocka skaliranja opisana v knjigi [27].

razporejevalnika bremena pošljejo na prileţne instance. V nadaljevanju tega poglavja so te tri osi skaliranja predstavljene še podrobneje.

Pri horizontalnem skaliranju se izvaja več identičnih instanc za razporejevalnikom bremena, ki enakomerno med vse instance razporeja breme oz. zahteve uporabnikov. Podvajanje instanc je preprost način skaliranja uporaben tako za mikrostoritve kot tudi za večje monolitne aplikacije. Skaliranje po osi x je sicer preprosto, vendar se pojavljajo težave pri napovedovanju števila potrebnih vzporednih instanc. V infrastrukturi, ki ne omogoča samodejnega skaliranja, se tipično določi minimalno število vzporednih instanc, ki pa jih samodejno povečujemo glede na že opaţene trende v preteklosti in ročno v odvisnosti od trenutne odzivnosti ob nepričakovanih povišanih obremenitvah.

Ustvarjanje več instanc velikih monolitnih aplikacij za dosego višje razpoloţljivosti ali zmogljivosti je preprosto, vendar tipično ne potrebujemo več instanc celotne aplikacije, ampak samo nekaterih delov. Mikrostoritve so rezultat funkcionalne delitve večjih monolitnih aplikacij in predstavljajo skaliranje po osi y, kot je opisano v delih [9, 14, 15, 27]. Dekompozicija monolitnih arhitektur je obširen problem in zanj ne obstaja univerzalni način za določanje obsega posameznih mikrostoritev. Tipično ţelimo mikrostoritve, ki zajemajo neko funkcionalno celoto in niso pregrobe, kar ne prinaša prednosti mikrostoritev,

ali prefine, kar vnaša preveliko režijskega dela.

Os z je zadnja ortogonalna os kocke skaliranja, ki predstavlja skaliranje storitev glede na delitev podatkov ali uporabnikov. Podobno kot pri skaliranju po osi x se tudi tukaj izvajajo identične instance storitev. Podmnožice zahtevkov se preko razporejevalnika bremena pošljejo na zadolžene instance storitev. Posamezne instance so torej zadolžene za rokovanje s podmnožico zahtevkov, ki se tipično izberejo glede na uporabniško ime, tip uporabnika, primarni ključ entitete ipd. Skaliranje po osi z ni tipično za arhitekturo mikrostoritev, ampak je prisotno v arhitekturah namenskih aplikacij.

3.6 Skaliranje mikrostoritev v oblaku

Oblačne infrastrukture PaaS ponujajo platformo za razvoj, izvajanje, razhroščevanje, objavljanje in upravljanje aplikacij [28], brez dodatnega upravljanja z infrastrukturo, ki je tipično povezana z razvojem in objavo aplikacij. Infrastrukture PaaS omogočajo tudi naprednejše funkcije, kot so samodejno skaliranje, zaganjanje ter izklapljanje instanc storitev, razporejevalniki bremena (angl. Load balancer), registri storitev (angl. Service registry) in prehodi API (angl. API gateway). Poleg naprednejših funkcij imajo ponudniki oblačnih infrastruktur na voljo obsežno dokumentacijo za uporabo infrastrukture in vedno dosegljivo tehnično podporo za stranke. Oblačnih infrastruktur in ponudnikov le-teh je ogromno npr. Google App Engine, Heroku, IBM Bluemix, Microsoft Azure Web Sites, Oracle Cloud, SAP. Vse te oblačne infrastrukture PaaS so primerne za objavo in skaliranje mikrostoritev.

Arhitektura mikrostoritev omogoča izrabo naprednih funkcionalnosti in prednosti oblačnih infrastruktur PaaS. Ena izmed najbolj zanimivih funkcij oblačnih struktur je samodejno skaliranje mikrostoritev. Samodejno ustvarjanje in izklapljanje instanc mikrostoritev glede na trenutno breme omogoča učinkovito izrabo razpoložljivih virov, ki nam pri uporabi tuje oblačne infrastrukture močno zmanjša stroške in potreben administrativni čas.

Razporejevalnik bremena je pomemben del arhitekture mikrostoritev in skaliranja po osi x [9, 15], saj omogoča enakomerno razporejanje in posredovanje zahtevkov na posamezne instance mikrostoritev, ki navadno tečejo na več strojih na različnih lokacijah. Razporejevalnik bremena lahko omogoča tudi predpomnenje, nadzor dostopa, spremljanje in pregled dostopov do vmesnikov API. Razporejevalnik bremena je tipično dostopen na neki fiksni lokaciji bodisi kot del strojne opreme bodisi kot preprost programski strežnik proxy (angl. Proxy server) in omogoča dodajanje ter izklapljanje instanc mikrostoritev na način, ki je transparenten za vse odjemalce mikrostoritve.

Pri izvajanju mikrostoritev v oblaku se hitro poraja vprašanje, kje se nahajajo posa-

mezne instance [9]. Razporejevalniki bremena so navadno na statičnih naslovih IP (angl. Internet Protocol address), medtem ko so naslovi posameznih instanc mikrostoritev dinamično dodeljeni in se stalno spreminjajo. Obstaja več načinov za odkrivanje in vodenje evidence naslovov mikrostoritev, v nadaljevanju je opisan trenutno najaktualnejši način. Razporejevalniki bremena lahko za odkrivanje naslovov instanc mikrostoritev uporabljajo register storitev, v katerega mikrostoritve same objavljajo svoje lokacije in periodično obnavljajo svoj status. Na ta način lahko razporejevalnik bremena s poizvedbo o lokacijah instanc mikrostoritev pridobi aktualne naslove in vrata instanc, ter nato posreduje zahtevke. Prednost tega načina odkrivanja mikrostoritev je neopaznost s strani odjemalcev storitve. Register storitev je navadno vgrajeni del razporejevalnikov bremena v oblaku, do katerega instance dostopajo preko odjemalcev registra, kot so Zookeper[1], Consul[2], Netflix Eureka[3] itd. Register storitev je torej podatkovna zbirka aktualnih dinamično dodeljenih naslovov IP instanc mikrostoritev.

Prehod API je še ena napredna funkcionalnost oblačnih infrastruktur, ki predstavlja vstopno točko v sistem [9, 14, 15]. Prehod API izpostavlja vmesnike API mikrostoritev in omogoča predpomnenje, avtentikacijo, spremljanje prometa, združevanje odgovorov in se obnaša kot razporejevalnik bremena. Prehodi posredujejo uporabniške zahtevke na instance mikrostoritev in združujejo njihove odgovore. Vsak tip odjemalca (odjemalec REST, vmesnik UI, mobilna aplikacija) navadno uporablja svoj prehod, zaradi različnih potreb po informacijah z vmesnikov API. Glavna prednost prehodov je ovijanje strukture aplikacije, vendar na račun še ene visoko razpoložljive in upravljane komponente.

Združevanje mikrostoritev v virtualne stroje (angl. Virtual machine) pomeni veliko zasedenega režijskega prostora v oblaku. Mikrostoritve skupaj s potrebnimi knjižnicami in odvisnostmi lahko združujemo tudi v vsebnike (angl. Container), kot je recimo Docker[4], ki zasedejo mnogo manj prostora, ker si vsebniki delijo operacijski sistem in ne vsebujejo lastnega operacijskega sistema kot virtualni stroji. Uporaba vsebnikov za objavo mikrostoritev je opisana v več delih [11, 12, 13, 15] in kaže na vsesplošno uporabo tega pristopa. Vsebnike tipično razpršimo na več oblačnih virtualnih strojev, saj to zagotavlja visoko razpoložljivost tudi v primeru odpovedi posameznih strojev.

[1] `https://zookeeper.apache.org/`
[2] `https://www.consul.io/`
[3] `https://www.github.com/Netflix/eureka`
[4] `https://www.docker.com/`

3.7 Ogrodje KumuluzEE

Arhitektura mikrostoritev z razgradnjo aplikacije na več manjših storitev omogoča visoko skalabilnost, vendar hkrati prinaša tudi težave oz. pomanjkljivosti. Ena izmed glavnih pomanjkljivosti mikrostoritev je obseg dodatnega dela z združevanjem v vsebnike, dodajanjem odvisnosti, konfiguracijo vsebnikov in objavo na strežnik ali v oblačno infrastrukturo. Dodatno režijsko delo lahko poleg implementacije storitev predstavlja velik del porabljenega časa v fazi razvoja mikrostoritve. Pri naslavljanju dodatnega režijskega dela so nam v pomoč ogrodja, ki omogočajo samodejno konfiguracijo in združevanje projektov v skladu s koncepti mikrostoritev. Eno izmed takšnih ogrodij za platformo Java EE je ogrodje KumuluzEE [11, 17], ki omogoča modularno izbiro potrebnih komponent Java EE s pomočjo orodja Maven[5].

Platforma Java EE ne omogoča preprostega načina konfiguracije in poganjanja aplikacij kot samostojnih, zato se v tej tehnologiji navadno razvija monolitne aplikacije in jih objavlja na aplikacijske strežnike. V ta namen je bilo razvito ogrodje KumuluzEE, ki je podrobneje opisano v delu [17]. To ogrodje omogoča uporabo tehnologij platforme Java EE za razvoj mikrostoritev brez dodatnega lastnega vključevanja in konfiguracije, ter s tem lajša dolgotrajen in težaven proces konfiguracije aplikacij. Ogrodje omogoča tudi samodejno združevanje celotne mikrostoritve in potrebnih komponent v arhiv JAR, ki ga lahko izvajamo v modernih oblačnih infrastrukturah ali v lastni infrastrukturi.

Ogrodje KumuluzEE še ne omogoča podpore čisto vseh komponent specifikacije Java EE [17], kljub temu pa omogoča razvoj naprednih mikrostoritev z izpostavljenimi vmesniki API. V ogrodju so trenutno na voljo komponente Servlet 3.1, WebSocket 1.1, JSP 2.3, EL 3.0, CDI 1.2, JPA 2.1, JAX-RS 2.0, JSF 2.2, Bean Validation 1.1 in JSON-P 1.0. Avtorji imajo v prihodnosti namen dodati še ostale komponente platforme Java EE in tudi alternativne implementacije teh tehnologij, kar bi omogočalo celosten razvoj aplikacij, ki temeljijo na platformi JAVA EE.

[5] `https://maven.apache.org/`

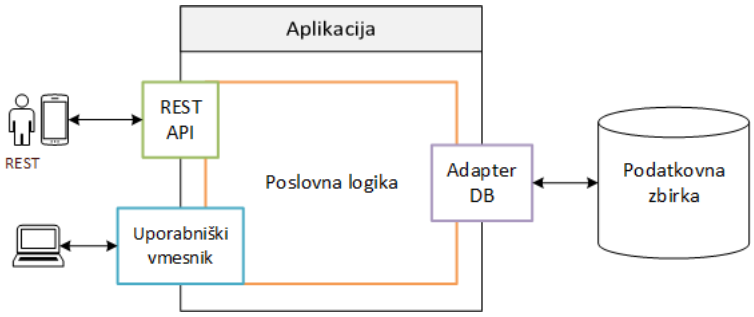
Poglavje 4

Zasnova arhitekture

Novo aplikacijo tipično implementiramo na podlagi monolitne arhitekture, kjer so vse komponente aplikacije razvite, združene in objavljene na strežnik skupaj v paketu. Razvoj monolitne aplikacije je preprost in nam sprva zadošča v smislu razpoložljivosti in zmogljivosti. Z večanjem števila uporabnikov skaliramo najprej z ustvarjanjem več vzporednih instanc in nato nadaljujemo s funkcionalno dekompozicijo arhitekture na mikrostoritve ter ločevanjem uporabniških zahtevkov, kot je že opisano v poglavju 3.5. To poglavje v nadaljevanju opisuje in prikazuje razvoj vse od monolitne arhitekture do visoko skalabilne arhitekture mikrostoritev s sprotnim dodajanjem naprednih oblačnih struktur iz poglavja 3.6. Zasnova in implementacija arhitekture mikrostoritev na začetku razvoja aplikacije skrajša razvojni cikel in omogoča elastično skalabilnost od prve objave na strežnik ali v oblak.

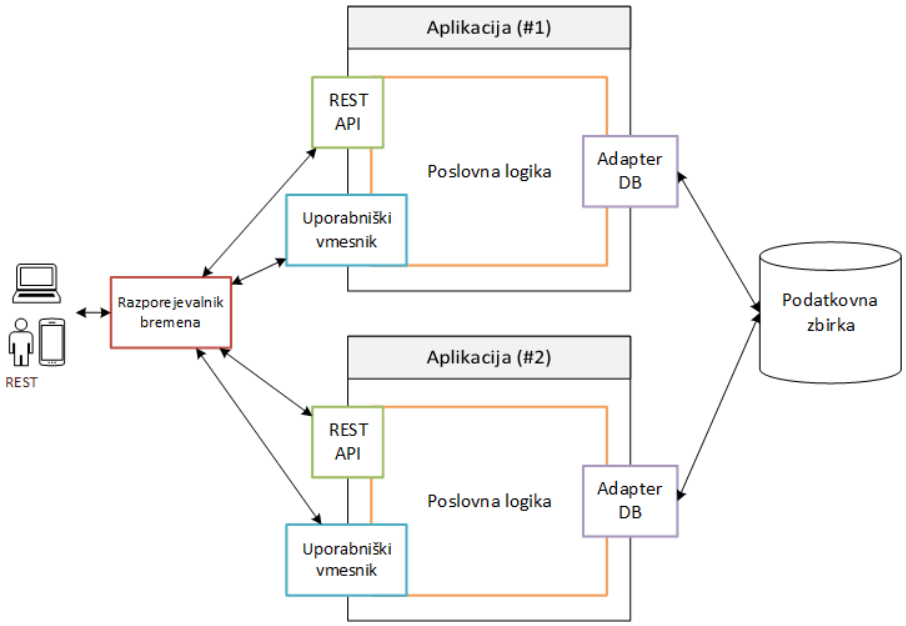
4.1 Monolitna arhitektura

Na sliki 4.1 je prikazana preprosta monolitna aplikacija, jedro katere predstavlja poslovna logika, ki se preko vmesnikov povezuje z zunanjim svetom. Aplikacija izpostavlja vmesnik API tipa REST in uporabniški vmesnik, ter se preko adapterja povezuje na podatkovno zbirko. Ta aplikacija predstavlja primer celostne monolitne aplikacije, ki bi jo združili in objavili na strežnik ali v oblak. Trenutno je takšen način razvoja aplikacij najbolj razširjen in večinoma tudi zadošča potrebam manjših organizacij in uporabnikom aplikacij. Kadar imamo veliko uporabnikov in večja bremena lahko monolitno aplikacijo v prvi fazi vertikalno skaliramo s povečevanjem zmogljivosti infrastrukture ali pa v nadaljevanju horizontalno skaliramo z ustvarjanjem več instanc.



Slika 4.1: Monolitna aplikacija.

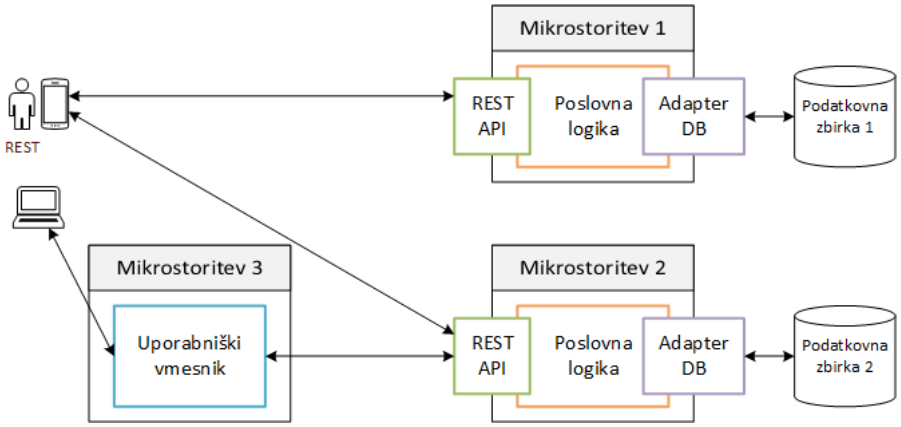
Primer horizontalnega skaliranja monolitne aplikacije z dvema instancama je prikazan na sliki 4.2. Pri horizontalnem skaliranju ustvarimo več vzporednih instanc aplikacije in jih objavimo za razporejevalnik bremena. Zmogljivost in potrebe po virih rastejo sorazmerno s številom instanc aplikacije. Več instanc aplikacije pa ne zagotavlja zgolj višje zmogljivosti in odzivnosti, temveč tudi višjo razpoložljivost v primeru napak. Napaka v aplikaciji ali infrastrukturi tako ne zruši celotne aplikacije ampak samo eno instanco, katere breme se enakomerno porazdeli med ostale izvajajoče se instance.



Slika 4.2: Horizontalno skaliranje.

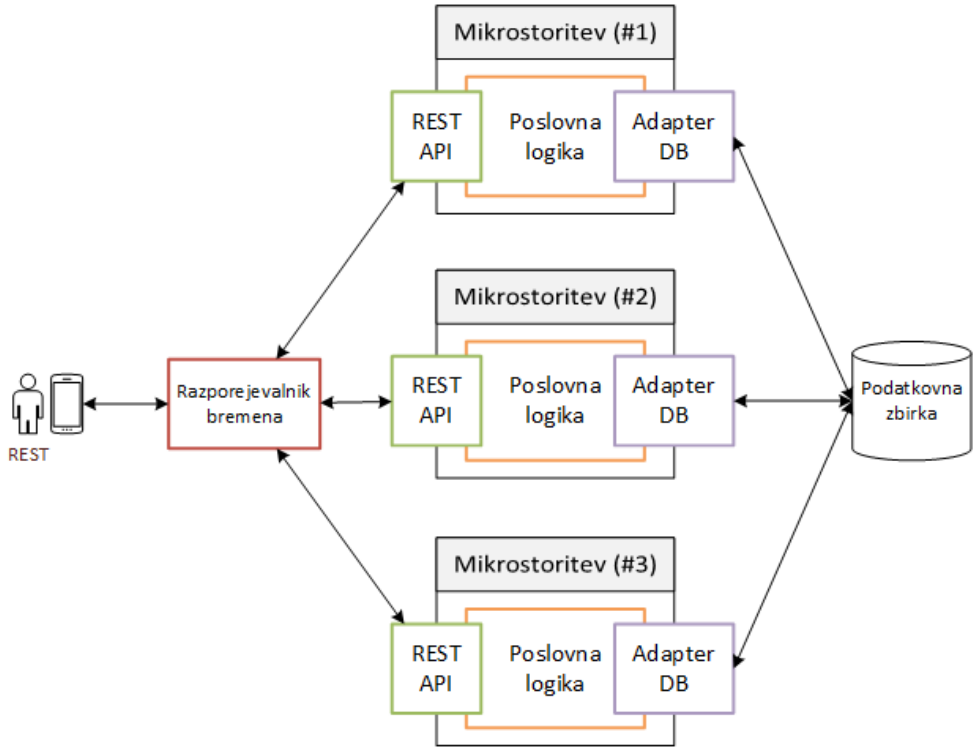
4.2 Arhitektura mikrostoritev

Skaliranje monolitnih aplikacij zagotavlja visoko razpoložljivost in zmogljivost, vendar na račun velike porabe virov in ogromnih stroškov. Navadno ni potrebno skaliranje celotne aplikacije, temveč le posameznih komponent in tukaj je razvidna potreba po razbitju velikih monolitnih aplikacij. Funkcionalna dekompozicija monolitne aplikacije na mikrostoritve je prikazana na sliki 4.3. Na tej sliki je prikazano razbitje velike monolitne aplikacije na uporabniški vmesnik in več mikrostoritev, ki zajemajo posamezne funkcionalne celote. Posamezne mikrostoritve so podobne monolitni aplikaciji s poslovno logiko v osrčju in izpostavljenimi vmesniki API tipa REST, vendar z ločenimi manjšimi podatkovnimi zbirkami. Odjemalci in mikrostoritve se na mikrostoritve povezujejo preko izpostavljenih vmesnikov API in s tem dosegajo šibko sklopljenost. S skaliranjem po osi y smo dosegli funkcionalno razbitje, ki nam je v nadaljevanju snovanja arhitekture omogočala skalabilnost posameznih mikrostoritev.



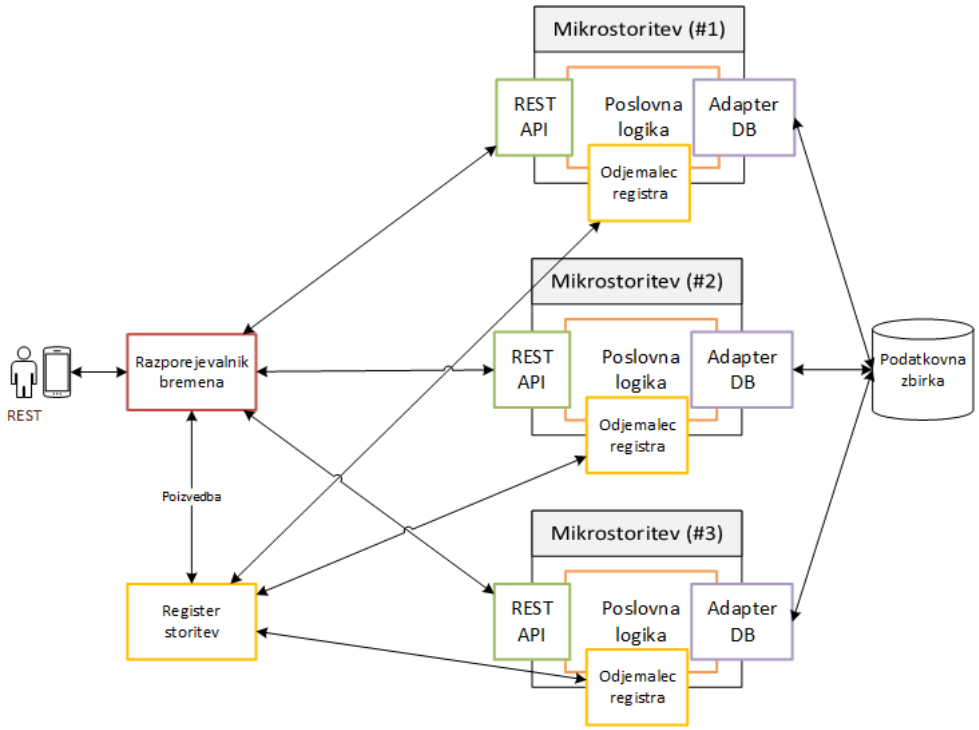
Slika 4.3: Funkcionalna dekompozicija na mikrostoritve.

Posamezne mikrostoritve lahko tako kot monolitne aplikacije skaliramo horizontalno z ustvarjanjem več vzporednih instanc, kot je prikazano na sliki 4.4. Več instanc objavimo za razporejevalnik bremena, ki skrbi za enakomerno razporejanje bremena med posamezne instance tudi v primeru izpadov in napak. Instance mikrostoritev skupaj s potrebnimi knjižnicami in odvisnostmi navadno tečejo v svojih vsebnikih na več virtualnih strojih v oblaku, kar zmanjša režijski prostor v infrastrukturi. S horizontalnim skaliranjem smo tako zagotovili visoko razpoložljivost in zmogljivost, na drugi osi pa smo s funkcionalno dekompozicijo omogočili skaliranje potrebnih mikrostoritev in zmanjšali porabo virov.



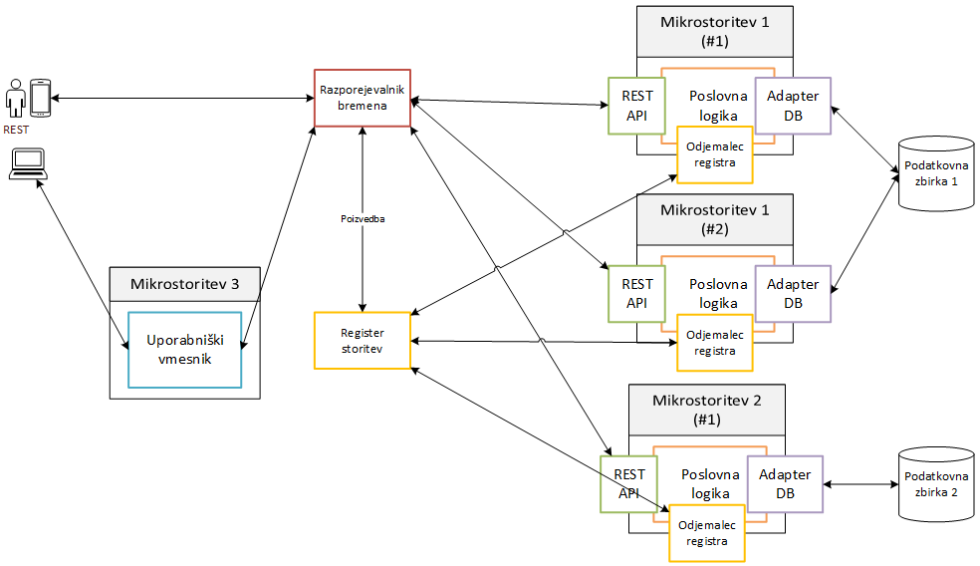
Slika 4.4: Horizontalno skaliranje mikrostoritev.

Instance aplikacij, ki se izvajajo na tradicionalnih strežnikih, imajo navadno statične naslove IP, kar poenostavi administrativno določanje naslovov instanc v razporejevalniku bremena. V oblačnih infrastrukturah, kjer se naslovi instanc aplikacij ne le dinamično spreminjajo, temveč se število instanc tudi stalno spreminja, prihaja do vprašanja o lokacijah instanc aplikacij. V oblačnih infrastrukturah je zato potrebno dinamično osveževanje naslovov instanc aplikacije. To navadno rešujemo z registrom storitev na razporejevalniku bremena in odjemalci registra na posameznih instancah aplikacij oz. v našem primeru mikrostoritev, kot je prikazano na sliki 4.5. Instance mikrostoritev periodično osvežujejo svoj status in naslov v registru storitev. Zahtevki poslani s strani odjemalcev na razporejevalnik bremena, ki najprej poizvede o lokacijah instanc mikrostoritve, so posredovani na vmesnik API instance mikrostoritve, ki je aktivna in ima posodobljen naslov v registru storitev. Instance mikrostoritev, ki ne uspejo posodobiti svojega statusa ali naslova v registru storitev, se obravnavajo kot neaktivne.



Slika 4.5: Odkrivanje instanc mikrostoritev z registrom storitev.

Na sliki 4.6 je prikazana arhitektura mikrostoritev z dodanimi naprednimi oblačnimi strukturami, ki so nujne za izvajanje v oblačnih infrastrukturah. Na začetku je razvidno razbitje na mikrostoritve uporabniškega vmesnika in mikrostoritve, ki izpostavljajo vmesnik API tipa REST. Mikrostoritve, ki izpostavljajo uporabniški vmesnik, v oblak prav tako objavimo za razporejevalnik bremena in jim dodamo odjemalec registra, vendar smo shemo poenostavili in to izpustili. Na tej sliki je razvidno, da se zahtevki odjemalcev mikrostoritev pošljejo na razporejevalnik bremena, ki se navado nahaja na statičnem naslovu IP in tako omogoča preprosto naslavljanje. Razporejevalnik bremena na podlagi informacij iz registra storitve posreduje zahtevke na instance mikrostoritev in vrne odgovor odjemalcu. Instanca mikrostoritve poleg poslovne logike in raznih vmesnikov vsebuje še odjemalec registra in si deli podatkovno zbirko z drugimi instancami enake mikrostoritve. Podatkovne zbirke mikrostoritev pa so med sabo ločene in so lahko različnih tipov, kar omogoča prilagajanje posameznih podatkovnih zbirk glede na podatke in potrebe mikrostoritev.



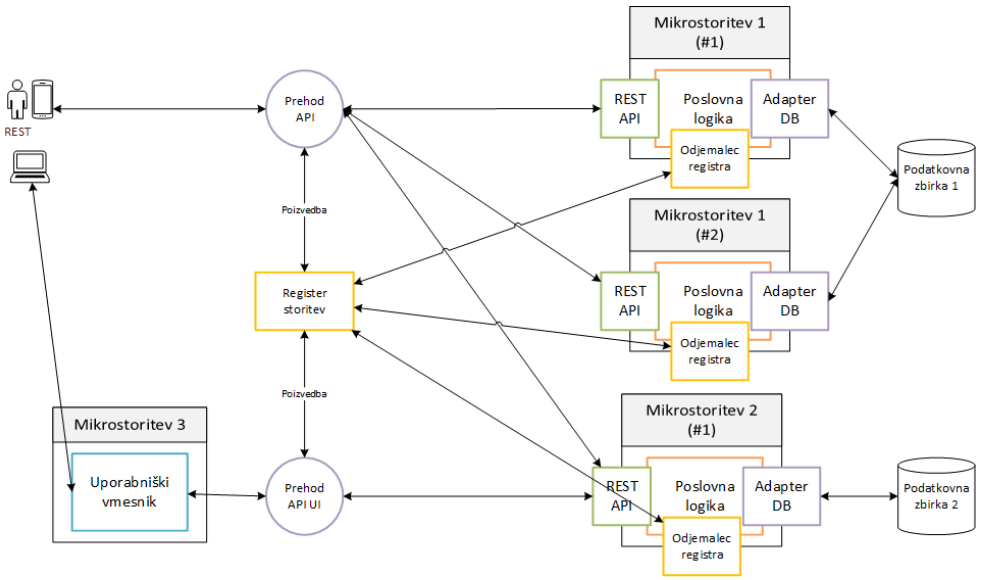
Slika 4.6: Arhitektura mikrostoritev.

4.3 Prehod API

Odjemalci lahko do mikrostoritev dostopajo preko razporejevalnikov bremena, vendar se funkcionalnost in število mikrostoritev stalno spreminja. Stalno spreminjanje mikrostoritev in izpostavljenih vmesnikov API ob neposrednem dostopu do razporejevalnikov bremena prinaša potrebo po spreminjanju uporabniškega vmesnika in mobilnih odjemalcev ali pa onemogoča proste spremembe po funkcionalnosti mikrostoritev. Arhitektura mikrostoritev naj bi omogočala stalno izboljševanje in nadgrajevanje mikrostoritev, zato je boljši način za dostop do mikrostoritev preko prehoda API, ki je prikazan na sliki 4.7. Ta nova vstopna točka v sistem, nam omogoča upravljanje z zahtevki in zbiranjem odgovorov mikrostoritev na enem mestu. Prehod API ima veliko funkcij, najpomembnejše so usmerjanje zahtevkov, združevanje rezultatov več mikrostoritev in rokovanje s spremembami mikrostoritev. Prehod API podobno kot razporejevalnik bremena izpostavlja vmesnike API mikrostoritev, vendar navadno izpostavlja tudi lastne vmesnike API, ki v nadaljevanju združujejo rezultate več različnih vmesnikov API v en skupen večji odgovor, kar omogoča manjše število zahtevkov s strani odjemalcev. Rokovanje s spremembami mikrostoritev in njihovih vmesnikov API pa omogoča nemoteno delovanje odjemalcev.

Odjemalci navadno uporabljajo različne mikrostoritve in za prikaz uporabniškega vmesnika potrebujejo druge informacije. Uporaba skupnega prehoda API za vse tipe odjemalcev tako ni najboljša rešitev, kar je razvidno predvsem pri težavni izolaciji uporabniških

vmesnikov [9] in nepotrebnih odvečnih informacijah pri vračanju dogovorov. Za vsak tip odjemalca navadno uporabimo lastni specifični prehod API, ki uporablja najmanjšo podmnožico mikrostoritev in vrača odgovore, ki vsebujejo le nujne informacije ter so čim krajši. Uporaba več prehodov API razbremeni le-te in zmanjša verjetnost po ustvarjanju ozkega grla. Ločevanje prehodov API glede na odjemalce prav tako olajša koordinacijo in razvoj mikrostoritev med razvojnimi skupinami, kjer lahko vsaka skupina skrbi za svoj lastni prehod API. Uporaba več prehodov API je prikazano na sliki 4.7 in še bolj nazorno na poenostavljeni sliki arhitekture 4.8.

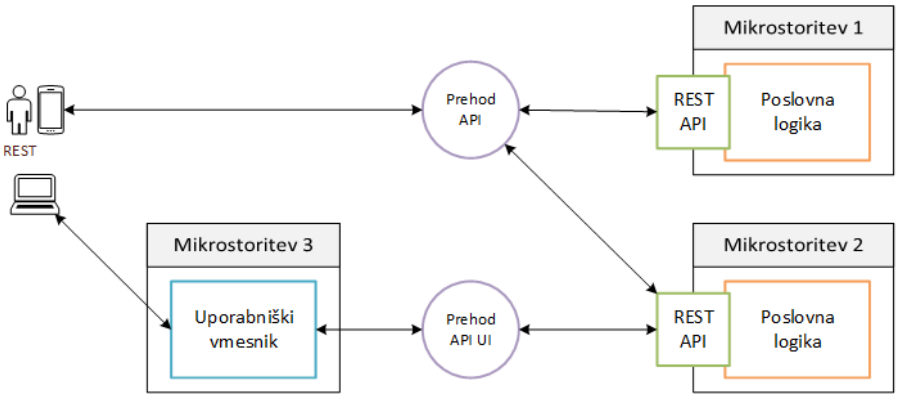


Slika 4.7: Arhitektura mikrostoritev z ločenimi prehodi API.

4.4 Poenostavljena arhitektura

Arhitektura mikrostoritev prinaša elastično skalabilnost posameznih mikrostoritev in rokovanje z velikim številom uporabnikov ter ogromnimi bremeni. Slika 4.8 prikazuje poenostavljeno arhitekturo mikrostoritev z vsemi potrebnimi strukturnimi elementi za objavo v oblačne infrastrukture. Posamezni odjemalci preko namenskih prehodov API dostopajo do izpostavljenih mikrostoritev. Instance mikrostoritev so objavljene za prehod API, ki se obnaša kot razporejevalnik bremena, in svoj status osvežujejo v registru storitev. Na podlagi informacij iz registra storitev so zahtevki odjemalcev preko prehoda API posredovani na vmesnike API aktivnih instanc mikrostoritev, kjer se nato izvede poslovna logika in po isti poti nazaj skozi infrastrukturo vrne odgovor mikrostoritve. Posamezne instance

mikrostoritve si med sabo delijo namensko podatkovno zbirko, ki pa je ločena od drugih zbirk mikrostoritev. Posamezne instance mikrostoritev lahko poljubno oz. samodejno skaliramo v oblaku in s tem zagotovimo zmogljivost in razpoložljivost.



Slika 4.8: Poenostavljena končna arhitektura mikrostoritev.

Poglavje 5

Implementacija arhitekture

Arhitektura mikrostoritev prinaša visoko skalabilnost v primerjavi z monolitno arhitekturo, kot je podrobneje opisano v poglavju 3. Implementacija posameznih mikrostoritev in njihove medsebojne komunikacije pa prinaša svojevrstne težave. V primeru slabe izvedbe in konfiguracije arhitekturnih elementov lahko izniči prednosti mikrostoritev, predvsem pri porabi virov v sodobnih oblačnih infrastrukturah. To poglavje v nadaljevanju opisuje in dokumentira implementacijo mikrostoritev s platformo Java EE in njihovo konfiguracijo z ogrodjem KumuluzEE. V poglavju je predstavljen razvoj dveh mikrostoritev vse od zasnove projekta z orodjem Maven pa do implementacije vmesnikov, ki izpostavljajo ključne operacije za delo z entitetami. Na kratko je predstavljena komunikacija znotraj arhitekture in zagon posameznih mikrostoritev. Na koncu pa so opisani razlogi za izbor oblačne infrastrukture Heroku[1] in objava mikrostoritev in podatkovnih zbirk v tej infrastrukturi PaaS. Postopek objave v vsebnike vsebuje več korakov, ki pa so preprosti in dobro opisani tudi v dokumentaciji infrastrukture Heroku.

5.1 Zasnova projekta

Za implementacijo arhitekture smo zasnovali preprost primer pridobivanja in vzdrževanja izdelkov ter akcij, ki bi se teoretično lahko uporabljali za namenske aplikacije trenutnih ugodnosti trgovcev. Trgovci bi skrbeli za dodajanje, urejanje ter brisanje izdelkov in urejanje aktualnih informacij o ugodnostih. Na podlagi definiranih območij delovanja pa bi lahko uporabniki aplikacij oz. kupci pridobivali informacije o trenutnih ugodnostih izdelkov. Trgovci bi lahko kupce o aktualnih ugodnostih obveščali preko namenskih aplikacij, informacije glede količine popusta in dejanske cene izdelkov pa bi pridobili samodejno v

[1] `https://www.heroku.com/`

bližnji okolici trgovine. Na podlagi aplikacij o osveščanju kupcev bi trgovci prihranili na tisku in razpošiljanju letakov, ter bi preprosto in hitro obvestili širšo množico kupcev.

Projekt smo ustvarili s pomočjo orodja Maven in vanj dodali štiri module, ki so prikazani v izseku 5.1. Projekt vsebuje modul za lokacijsko odvisnost, modul za skupne entitete mikrostoritev in dve mikrostoritvi, ki skrbita za pridobivanje ter urejanje izdelkov in akcij. Posamezni mikrostoritvi bi glede na prakse morali ločiti v lastna projekta, vendar smo vse module ustvarili znotraj enega projekta zaradi preprostosti, lažje izvedbe verzioniranja s pomočjo orodja Git in preproste konfiguracije s pomočjo ogrodja KumuluzEE. Ne glede na to, da so omenjeni mikrostoritvi skupaj v projektu, lahko s pomočjo ogrodja KumuluzEE storitvi objavimo ločeno vsako v svojem arhivu, ki je pripravljen za izvajanje v vsebniku Docker.

```
<modules>
    <module>lokacije</module>
    <module>entitete</module>
    <module>izdelki</module>
    <module>akcije</module>
</modules>
```

Izsek 5.1: Moduli projekta iz konfiguracijske datoteke pom.xml.

Modul za lokacijsko odvisnost omogoča definicijo lokacijsko odvisnih vmesnikov API tipa REST s pomočjo lastne anotacije razredov, ki implementirajo te vmesnike. Implementacija modula in anotacija je podrobneje predstavljena v poglavju 2. Sam modul je vključen v korenski projekt in ga je mogoče vključiti v druge module s podajanjem odvisnosti, kot smo to storili pri obeh modulih mikrostoritev v konfiguracijskih datotekah.

Modul za entitete vsebuje skupne entitete podatkovnega modela, ki se uporabljajo v poslovni logiki mikrostoritev in na vmesnikih API. V podatkovnem delu sta dve ključni entiteti izdelek in akcija, ki ju upravljata priležni mikrostoritvi. Modul je možno podobno kot modul za lokacijsko odvisnost vključiti v druge module s podajanjem odvisnosti v konfiguracijski datoteki.

Projekt vsebuje dva modula, ki implementirata vsak svojo neodvisno mikrostoritev, ki je pripravljena za izvajanje v oblačni infrastrukturi PaaS. Prvi modul implementira mikrostoritev za pridobivanje, urejanje in brisanje izdelkov. Drugi modul pa implementira mikrostoritev za pridobivanje, urejanje in brisanje akcij posameznih izdelkov prve mikrostoritve. Vsaka mikrostoritev ima svojo podatkovno zbirko za entiteto, ki jo prikazuje in upravlja, kar je skladno s koncepti mikrostoritev opisanimi v poglavju 3.

Zgoraj navedeni moduli in njihove implementacije so podrobneje opisane v naslednjih poglavjih, ki so ključna za poglobljeno razumevanje zasnovane arhitekture iz poglavja 4,

posameznih mikrostoritev in njihove medsebojne interakcije pri doseganju celostne funkcionalnosti sistema.

5.2 Definicija podatkovnega modela

Modul za entitete predstavlja skupne entitete podatkovnega modela, ki se uporabljajo v poslovni logiki implementiranih mikrostoritev in ga je možno poljubno razširiti z novimi entitetami za nadaljnje potrebe mikrostoritev. Trenutno modul vsebuje le dve entiteti izdelek in akcija, na podlagi katerih smo implementirali preprost primer arhitekture. Modul je možno preprosto vključiti v mikrostoritve s podajanjem odvisnosti v konfiguracijskih datotekah. Poleg entitet modul vsebuje informacije za povezavo na podatkovne zbirke mikrostoritev.

Konfiguracija modula za entitete je podana v datoteki pom.xml in vsebuje le dve odvisnosti, ki sta prikazani v izseku 5.2. Prva odvisnost podaja komponento JPA ogrodja KumuluEE, ki je implementirana z rešitvijo EclipseLink. Ta predstavlja standardno objektnorelacijsko rešitev in omogoča uporabo številnih naprednejših funkcij. Druga odvisnost pa podaja gonilnik za podatkovno zbirko PostgreSQL platforme Java, ki omogoča povezovanje na podatkovno zbirko na standarden in neodvisen način.

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-jpa-eclipselink</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4-1201-jdbc41</version>
</dependency>
```

Izsek 5.2: Odvisnosti modula za entitete.

V izseku 5.3 sta prikazana poenostavljena razreda, ki implementirata uporabljeni entiteti v mikrostoritvah. Ena izmed entitet je entiteta izdelek, ki se uporablja v obeh mikrostoritvah. Izdelek je enolično identificiran s poljem id ter podaja še informacijo o nazivu, opisu in ceni. Druga entiteta je akcija, ki pa se trenutno uporablja le v eni izmed mikrostoritev. Entiteta akcija je enolično identificirana s poljem id ter podaja še informacijo o veljavnosti akcije, količini popusta in podaja polje id izdelka na katerega se navezuje, kot tuji ključ. Obe entiteti izpostavljata metode za pridobivanje ter nastavljanje polj in

metodo za lepši izpis podatkov entitete.

```
@Entity
public class Izdelek {
    @Id
    private Integer id;
    private String naziv;
    private String opis;
    private double cena;
    ...
}

@Entity
public class Akcija {
    @Id
    private Integer id;
    private Date veljavnostOd;
    private Date veljavnostDo;
    private double popust;
    private Integer idIzdelka;
    ...
}
```

Izsek 5.3: Poenostavljen podatkovni model mikrostoritev.

V modulu za entitete so podani še podatki za dostop do podatkovnih zbirk mikrostoritev, v katerih se shranjujeta dve ključni entiteti. Vsaka entiteta se shranjuje v lastni podatkovni zbirki, ki je dostopna na svojem naslovu. Poleg naslova pa je za vsako podatkovno zbirko podan tudi tip gonilnika, ki je odvisen od tipa podatkovne zbirke, in prijavni podatki za dostop. Podatki so nujno potrebni za delovanje komponente JPA, ki poenostavlja delovanja in omogoča preprostejši razvoj poslovne logike.

5.3 Implementacija mikrostoritev

To poglavje opisuje dva modula, ki implementirata neodvisni mikrostoritvi za pridobivanje, urejanje in brisanje entitet. Prva mikrostoritev omogoča delo z izdelki, medtem ko druga mikrostoritev omogoča delo z akcijami. Mikrostoritvi sta implementirani s platformo Java EE, konfigurirani z ogrodjem KumulzEE in pripravljeni za izvajanje v oblačnih infrastrukturah. Obe mikrostoritvi izpostavljata preprost vmesnik API tipa REST in lahko služita

kot primer implementacije mikrostoritev s platformo Java EE.

```
<dependency>
    <groupId>si.fri.akcije</groupId>
    <artifactId>entitete</artifactId>
    <version>${akcije.version}</version>
</dependency>
<dependency>
    <groupId>si.fri.akcije</groupId>
    <artifactId>lokacije</artifactId>
    <version>${akcije.version}</version>
</dependency>

<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-core</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-servlet-jetty</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-jax-rs-jersey</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
```

Izsek 5.4: Odvisnosti mikrostoritev.

V izseku 5.4 so prikazane odvisnosti mikrostoritev iz konfiguracijske datoteke. Obe mikrostoritvi vsebujeta pet ključnih odvisnosti, ki so nujno potrebne za delovanje. Odvisnost na modul za entitete je potrebna za uporabo skupnih entitet iz podatkovnega modela. Modul za lokacije pa je potreben za uporabo lokacijsko odvisne anotacije na razredih, ki implementirajo vmesnike API tipa REST. Poleg dveh modulov, ki smo jih implementirali v projektu, pa so podane še tri odvisnosti komponent ogrodja KumuluzEE. Komponenta kumuluzee-core je nujno potrebna za konfiguracijo z ogrodjem KumuluzEE in dodaja možnost zaganjanje modula kot lastno aplikacijo platforme Java EE. Komponenta kumuluzee-servlet-jetty dodaja strežnik HTTP, ki bo tekom izvajanja aplikacija prestregal

zahtevke in jih naslavljal na komponente platforme Java EE. Komponenta kumuluzee-jax-rs-jersey pa omogoča izpostavljanje vmesnikov API tipa REST in uporabo nujno potrebnih anotacij za implementacijo teh vmesnikov. Sama implementacija vmesnikov API tipa REST je neodvisna od ogrodja KumuluzEE in tako ni potrebno nobeno dodatno znanje za uporabo le-tega.

V obeh modulih mikrostoritev smo v nadaljevanju implementirali razred z anotacijo @ApplicationPath, ki omogoča nastavljanje glavnega dela naslova URL za dostop do izpostavljenih virov. V naši implementaciji smo za glavni del naslova nastavili različico mikrostoritve, kar omogoča preprosto ugotavljanje različic naslovljenih instanc mikrostoritev iz zahtevkov REST. V modulih mikrostoritev pa smo dodali še razrede, ki implementirajo vmesnike API tipa REST in poslovno logiko potrebno za osnovno delovanje mikrostoritev. Pri osnovnem delovanju so mišljene funkcionalnosti za pridobivanje, dodajanje in urejanje entitet v podatkovnih zbirkah. Na razreda mikrostoritev, ki implementirata vmesnika, pa smo dodali še anotacijo za lokacijsko odvisnost.

V razredih, ki implementirajo vmesnike API, smo implementirali več metod, ki sprejemajo zahtevke in vračajo odgovore tipa JSON. Vmesniki mikrostoritev izpostavljajo metode REST, ki vračajo entitete in omogočajo njihovo urejanje. Na voljo so metode HEAD, GET, DELETE, POST in PUT. Obe mikrostoritvi izpostavljata metodo HEAD za preverjanje obstoja entitet, metodo GET za pridobivanje entitet in še eno metodo GET z drugim naslovom URL za pridobivanje vseh entitet. Na podlagi opisanih metod je mogoče preprosto preveriti obstoj in pridobiti podrobnejše informacije o entitetah izdelek in akcija. Entiteti izdelek ter akcija je možno dodati z uporabo metode POST, urejati z uporabo metode PUT in izbrisati z uporabo metode DELETE. Mikrostoritev pri dodajanju in spreminjanju posameznih akcij proži dodatne zahtevke HEAD na mikrostoritev za izdelke. To preverjanje ustreznosti polja ID pa predstavlja medsebojno komuniciranje implementiranih mikrostoritev. Metode poleg pričakovanih odgovorov vračajo tudi napake, ki so posledica napačnih odjemalčevih zahtevkov, in kode stanj družin 2xx, 4xx in 5xx. Vse implementirane metode skupaj omogočajo celostno upravljanje z entitetama izdelek in akcija. Na tem mestu je vredno ponovno omeniti, da vsaka mikrostoritev omogoča pridobivanje in urejanje le ene entitete, kar se sklada z arhitekturo mikrostoritev opisano v poglavju 3.

Izsek 2.4 prikazuje način podajanja več območij delovanja vmesnikov API. Obe mikrostoritvi vsebujeta podobni konfiguracijski datoteki za definicijo območij delovanja in skupaj z anotacijo za lokacijsko odvisnost na vmesnikih API vnašata lokacijsko odvisnost v delovanje mikrostoritev. Mikrostoritvi sta tako izpostavljeni preko lokacijsko odvisnih vmesnikov API, ki določajo pravice dostopa in prilagajajo delovanje storitve glede na lokacijo odjemalcev.

5.4 Zagon mikrostoritev

Po zasnovi projekta z orodjem Maven, konfiguraciji z ogrodjem KumuluzEE in implementaciji s platformo Java EE je potrebno posamezne mikrostoritve združiti v lastne arhive in jih objaviti v oblačne strukture PaaS. To poglavje v nadaljevanju prikazuje združevanje mikrostoritev v arhive in zagon mikrostoritev s pomočjo strežnika Jetty za razvojno (lokalno) testiranje brez naprednih funkcij oblačnih struktur. Pred zagonom mikrostoritev pa je potrebno vzpostaviti povezave na ločene podatkovne zbirke, ki vsebujejo ključne entitete podatkovnega modela. Zagotoviti je potrebno, da so podatkovne zbirke dostopne in polja konfiguracijske datoteke persistence.xml pravilno nastavljena glede na dostopne podatke podatkovnih zbirk.

Mikrostoritve je potrebno pred objavo v oblačne strukture združiti s pomočjo orodja Maven. To storimo z ukazom maven package, ki vzame prevedeno kodo in jo združi v arhiv. Ta arhiv je zdaj možno zagnati s preprostim ukazom platforme Java, v kolikor smo v sklopu konfiguracije ogrodja KumuluzEE dodali tudi preprost aplikacijski strežnik. Mikrostoritev lahko poženemo z ukazoma, ki sta prikazana v izsekih 5.5 in 5.6. V našem primeru bo aplikacijski strežnik Jetty izpostavil vmesnike API tipa REST in tako bo možno dostopati do implementirane mikrostoritve.

```
> java -classpath izdelki/target/classes;izdelki/target/
    dependency/* com.kumuluz.ee.EeApplication
```

Izsek 5.5: Zagon mikrostoritve v konzoli operacijskega sistema Windows.

```
> java -cp izdelki/target/classes:izdelki/target/dependency/*
    com.kumuluz.ee.EeApplication
```

Izsek 5.6: Zagon mikrostoritve v konzoli operacijskega sistema Linux.

V tem poglavju smo pokazali, kako preprosta sta konfiguracija in zagon mikrostoritve z ogrodjem KumuluzEE. Po vzpostavitvi strežnika Jetty in izpostavitvi mikrostoritve preko implementiranega vmesnika API je možno do mikrostoritve dostopati lokalno preko vrat 8080, kadar uporabljamo privzete nastavitve. Drugi del naslova URL pa je določen z delnim naslovom aplikacije, ki ga določa razred anotiran z anotacijo @ApplicationPath, in poti posameznih metod REST na samem vmesniku.

5.5 Objava v oblačno infrastrukturo PaaS

Oblačne infrastrukture različnih ponudnikov se razlikujejo glede na ponujene funkcionalnosti, cene in še mnoge druge dejavnike. Oblačna infrastruktura Heroku ponuja napredne

funkcionalnosti iz poglavja 3, brezplačen osnovni račun, ki omogoča objavo aplikacij in gostovanje podatkovnih zbirk, in celostno dokumentacijo funkcionalnosti ter uporabe le-teh. Prav zaradi vseh teh razlogov smo mikrostoritvi in podatkovni zbirki objavili v oblačno infrastrukturo Heroku, kar je podrobneje opisano v tem poglavju. V poglavju so na kratko opisana potrebna orodja za objavo v infrastrukturo Heroku in elementi infrastrukture. V nadaljevanju pa je opisan še dejanski potek konfiguracije ter objave v oblačno infrastrukturo kot tudi preprost način skaliranja instanc mikrostoritev.

Pred objavo mikrostoritve implementirane s platformo Java v oblačno infrastrukturo Heroku je potrebno poleg računa Heroku, platforme Java in orodja Maven uporabiti konzolno orodje Heroku Toolbelt, ki omogoča upravljanje in skaliranje aplikacij. S tem orodjem smo v nadaljevanju naložili obe mikrostoritvi v vsebnika Dyno. Vsebniki Dyno so preprosti vsebniki z operacijskim sistemom Linux namenjeni izvajanju enega uporabniškega ukaza v privzetem okolju, ki je odvisen od platforme aplikacije. Vsi vsebniki pa se nahajajo za prehodom API, ki omogoča izpostavitev funkcionalnosti mikrostoritev, samodejno skaliranje, razporejanje bremena, register storitev in predpomnenje.

Sistematični pristop k objavi v oblačno infrastrukturo omogoča preprosto objavo mikrostoritev v oblak brez dodatnih težav zaradi morebitnih odsotnosti virov. Pri objavi v oblak smo zato najprej ustvarili dve podatkovni zbirki, ki so nujni za uspešen zagon mikrostoritev, za vsako izmed mikrostoritev posebej. Ustvarjanje podatkovnih zbirk in konfiguracija prijavnih podatkov v oblačni infrastrukturi Heroku je možno izvesti kar preko spletnega vmesnika in je zelo preprosta. Po uspešni objavi podatkovnih zbirk smo za vsako pridobili naslov URL in prijavne podatke za dostop. Na podlagi teh podatkov smo nastavili konfiguracijsko datoteko persistance.xml, katere glavni del je prikazan v izseku 5.7.

```
<properties>
    <property name="javax.persistence.jdbc.driver" value="org.
        postgresql.Driver" />
    <property name="javax.persistence.jdbc.url" value="
        jdbc:postgresql://postgres.heroku.com:5432/database" />
    <property name="javax.persistence.jdbc.user" value="user" />
    <property name="javax.persistence.jdbc.password" value="
        password" />
</properties>
```

Izsek 5.7: Konfiguracija povezave na podatkovno zbirko.

Pred objavo mikrostoritev v oblačno infrastrukturo Heroku je potrebno vsaki mikrostoritvi dodati še ukaz za zagon v datoteko Procfile. Uporabljen ukaz za zagon mikrostoritve za delo z izdelki je razviden v izseku 5.6, ukaz za zagon mikrostoritve za delo z akcijami

pa je skoraj identičen.

Izsek 5.8 prikazuje zaporedje ukazov za objavo mikrostoritve za delo z izdelki v oblačno infrastrukturo z orodjem Heroku Toolbelt. Izsek prikazuje ukaz za inicializacijo nove aplikacije, ki omogoča objavo programske kode in pregled v nadzorni plošči. Drugi ukaz prikazuje objavo in združevanje celotne programske kode na glavni veji razvoja v repozitorij novo ustvarjene aplikacije. Zadnji ukaz pa prikazuje ustvarjanje instance mikrostoritve, ki se bo odzivala na zahtevke naslovljene na izpostavljeni naslov prehoda API, ki med drugim skrbi tudi za razporejanje bremena med trenutno delujočimi instancami mikrostoritve.

```
> heroku create izdelki
> git push heroku master
> heroku ps:scale web=1
```

Izsek 5.8: Objava mikrostoritve z orodjem Heroku Toolbelt.

Podobno kot mikrostoritev za delo z izdelki smo v oblačno strukturo objavili tudi mikrostoritev za delo z akcijami. Uporabili smo zaporedje ukazov z izseka 5.8 z izjemo poimenovanja nove aplikacije. Mikrostoritev za delo z akcijami komunicira z mikrostoritvijo za delo z izdelki, zato je bilo treba dodati tudi spremenljivko, ki predstavlja naslov URL vmesnika API. Spremenljivko naslov URL izdelkov smo dodali kar preko spletnega vmesnika v zavihku konfiguracijske spremenljivke.

Orodje Heroku Toolbelt omogoča preprost način skaliranja mikrostoritev z ustvarjanjem in povečevanjem instanc, kar je prikazano na izseku 5.9. To orodje omogoča način skaliranja na podlagi eksplicitnega dodajanja instanc, kar pa ne ustreza principu samodejnega skaliranja mikrostoritev, ki je ena izmed zahtev te arhitekture.

```
> heroku ps:scale web=2
> heroku ps:scale web+1
```

Izsek 5.9: Skaliranje mikrostoritve z orodjem Heroku Toolbelt.

Po objavi v oblačno infrastrukturo PaaS ponudnika Heroku smo se nato lotili verificiranja dveh osnovnih zahtev. Prva zahteva je zmožnost dodajanja lokacijske odvisnosti vmesnika API tipa REST z različnim obnašanjem glede na lokacijo odjemalca. Druga zahteva pa je zmožnost samodejnega skaliranja mikrostoritev ob manjših in večjih obremenitvah. V naslednjih poglavjih sta povzeti verifikaciji obeh zahtev.

Poglavje 6

Verifikacija arhitekture

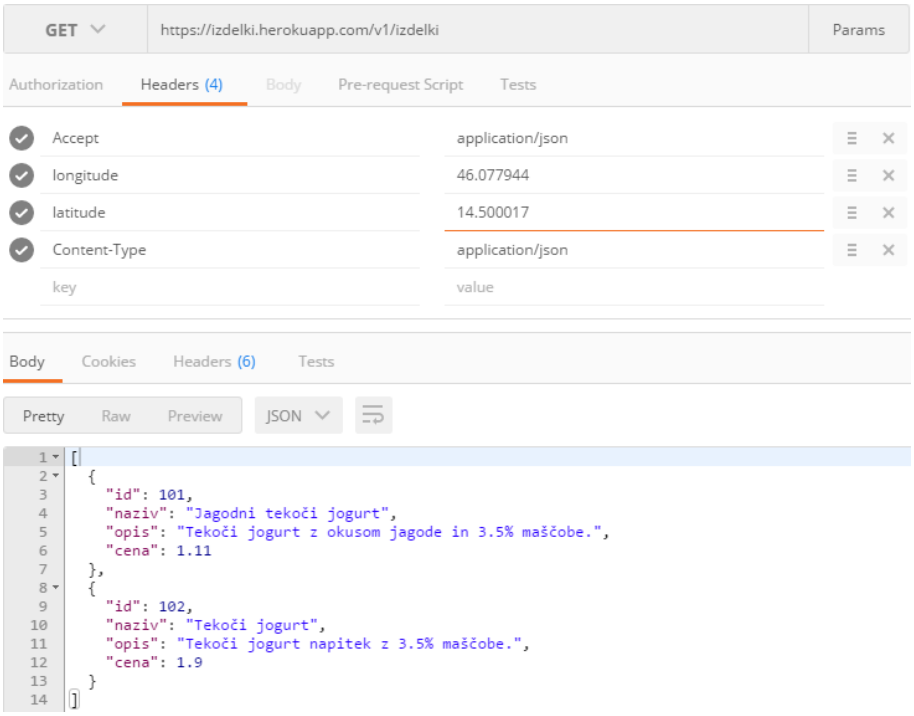
Po zasnovi in implementaciji arhitekture smo v sklopu magistrske naloge izvedli verifikacijo arhitekture in posameznih mikrostoritev. Celovito testiranje mikrostoritev je nujno za potrditev ustreznosti implementacije mikrostoritev in arhitekture kot celote glede na začetni zahtevi za lokacijsko odvisnost in samodejno skaliranje mikrostoritev. Na podlagi uspešnega verificiranja zahtev smo potrdili lastno implementacijo lokacijsko odvisne anotacije in konfiguracijo posameznih mikrostoritev z ogrodjem KumuluzEE. To poglavje v nadaljevanju podrobno opisuje verifikacijo lokacijske odvisnosti vmesnikov API tipa REST z različnim obnašanjem glede na lokacijo odjemalca in verifikacijo samodejnega skaliranja mikrostoritev. Na koncu poglavja pa je opisana verifikacija arhitekture glede na arhitekturne in načrtovalske vzorce.

6.1 Verifikacija lokacijske odvisnosti

V poglavju 2 je opisana definicija lokacijske odvisnosti vmesnikov API tipa REST in implementacija lastne anotacije za lažje dodajanje lokacijske odvisnosti na razrede implementirane s platformo Java EE. V tem poglavju pa je opisana verifikacija delovanja te anotacije glede na definirane zahteve, predvsem pri določanju pravic dostopa in prilagajanju delovanja mikrostoritve. Pri verifikaciji smo uporabili orodje Postman[1], ki se uporablja kot samostojni odjemalec storitev. To orodje omogoča pošiljanje zahtevkov z nastavljenimi atributi in pridobivanje zahtevkov različnih tipov. Samo orodje ponuja še množico naprednejših funkcij, vendar tekom verificiranja lokacijske odvisnosti niso bila potrebna.

Na sliki 6.1 je prikazan osnovni vmesnik orodja Postman za pošiljanje zahtevkov, ki

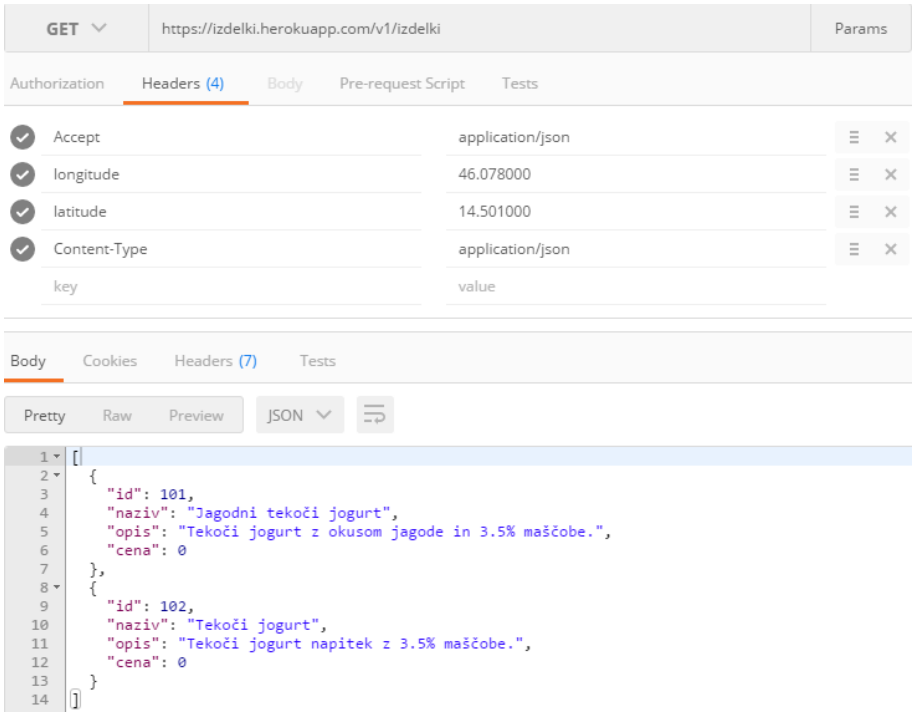
[1] `https://www.getpostman.com/`



Slika 6.1: Uspešen zahtevek in odgovor s kodo stanja 200.

omogoča celovito nastavljanje in urejanje zahtevkov. Na sliki je razviden primer zahtevka GET, ki je naslovljen na naslov URL vmesnika API, z nastavljenimi parametri glave, kot so tip sporočila in lokacija odjemalca. Na podlagi veljavne lokacije odjemalca in veljavnega zahtevka je v tem primeru mikrostoritev vrnila uspešen odgovor s kodo stanja 200, ki vsebuje krajši seznam izdelkov. Koda stanja 200 nam predstavlja uspešno izvršen zahtevek in omogoča preprosto programsko preverjanje uspešnosti na uporabniškem vmesniku.

Na sliki 6.2 je prikazan zahtevek GET in prilagojen odgovor mikrostoritve na podlagi lokacije odjemalca s kodo stanja 200. Na sliki 6.3 pa je prikazan zahtevek z neveljavno lokacijo odjemalca in odgovor mikrostoritve s kodo stanja 403, ki predstavlja zavrnjen zahtevek zaradi neuspešne avtorizacije. Oba zahtevka se od zahtevka s slike 6.1 razlikujeta le v nastavljeni zemljepisni dolžini in širini. Na podlagi teh treh zahtevkov in njihovih odgovorov je razvidna pravilnost implementacije lastne anotacije, ki omogoča preprosto dodajanje lokacijske odvisnosti vmesnika API tipa REST. Prilagojen odgovor mikrostoritve na podlagi lokacije odjemalca potrjuje zahtevo po prilagajanju delovanja mikrostoritve. Na podlagi odgovora z opisno informacijo o neveljavnosti odjemalčeve lokacije, pa smo potrdili zahtevo po določanju pravic dostopa do mikrostoritve. S pomočjo anotacije je tako mogoče preprosto določiti pravice dostopa odjemalcev glede na lokacijo odjemalcev



Slika 6.2: Uspešen zahtevek s prilagojenim odgovorom in kodo stanja 200.

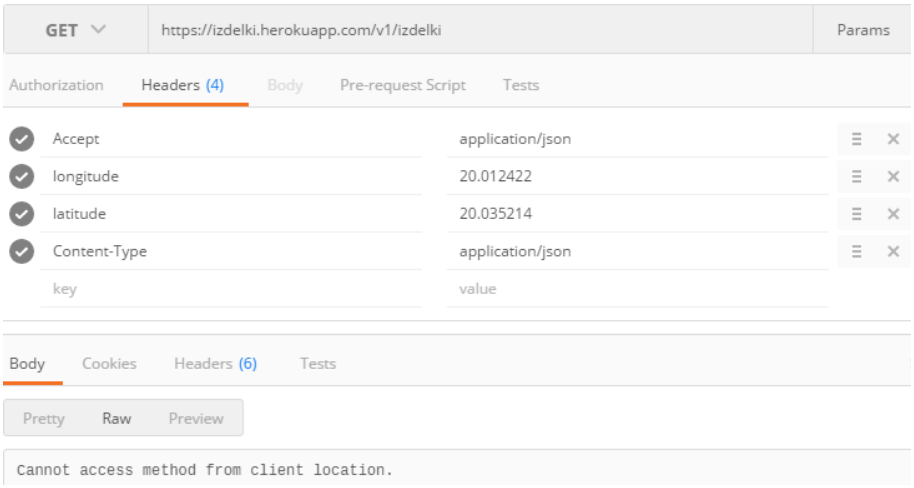
in prilagajati delovanje mikrostoritev ter njihovih odgovorov z možnostjo posredovanja nepopolnih odgovorov in zakrivanja informacij.

6.2 Verifikacija samodejnega skaliranja

Samodejno skaliranje po osi x z ustvarjanjem vzporednih instanc je eden izmed ključnih načinov skaliranja arhitekture mikrostoritev, ki je podrobneje opisan v poglavju 3. Samodejno skaliranje mikrostoritev je potrebno potrditi v fazi verifikacije arhitekture. V našem primeru smo samodejno skaliranje preverili s pomočjo oblačne infrastrukture PaaS ponudnika Heroku. Ta oblačna infrastruktura omogoča samodejno skaliranje mikrostoritev s pomočjo dodatka Adept Scale[2], ki omogoča samodejno ustvarjanje dodatnih vsebnikov Dyno, v katerih se nato izvajajo aplikacije. Celotno testiranje vključno z ustvarjanjem obremenitvenih testov in večje količine zahtevkov smo izvedli s pomočjo orodja Postman.

Dodatek Adept Scale omogoča spremljanje odzivnega časa in količine zahtevkov, ki so naslovljeni na posamezne mikrostoritve. Na podlagi preteklih odzivnih časov in dodatnih

[2] `https://elements.heroku.com/addons/adept-scale`

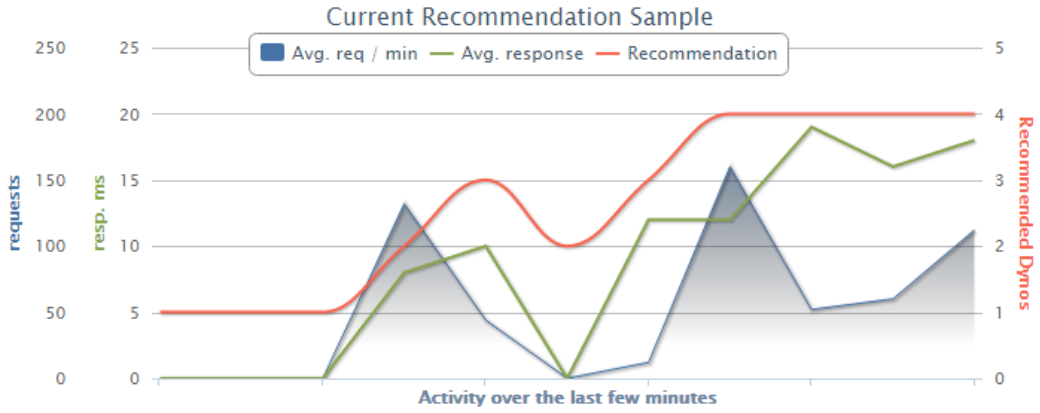


Slika 6.3: Neuspešen zahtevek in odgovor s kodo stanja 403.

ročnih nastavitev dodatek omogoča pametno dodajanje novih instanc v primeru večjih obremenitev ali izklapljanje odvečnih instanc v primeru običajnih obremenitev. Pred testiranjem samodejnega skaliranja smo nastavili parametre, kot sta hitrost odzivanja glede na obremenitve pri ustvarjanju in izklapljanju instanc. Poleg tega smo nastavili najmanjše število instanc mikrostoritev, ki so potrebne glede na pričakovane obremenitve, in najvišje število instanc, za katere smo še pripravljeni plačati njihovo vzporedno izvajanje.

Z orodjem Postman smo v načinu testiranja v nadaljevanju preizkusili delovanje samodejnega skaliranja mikrostoritev ob večjih obremenitvah v oblačni infrastrukturi Heroku. Orodje Postman omogoča ustvarjanje večje količine naključnih zahtevkov REST, tako zaporedno kot tudi vzporedno. Z orodjem Postman smo tako hkrati in dlje časa pošiljali zahtevke tipa HEAD, GET, POST, PUT ter DELETE, ki so obremenili instance mikrostoritev in celotno arhitekturo. Možnost ustvarjanja deloma naključnih zahtevkov REST znotraj orodja Postman pa ni obremenilo le instanc mikrostoritev, ampak tudi podatkovni zbirki. Na podlagi prometa in velike količine zahtevkov pa je dodatek Adept Scale celotni čas skaliral število instanc mikrostoritev glede na nastavljene in pridobljene parametre.

Na podlagi ustvarjenih zahtevkov in odzivnih časov mikrostoritev je dodatek Adept Scale ustvaril graf, ki je prikazan na sliki 6.4. Iz tega grafa je razvidna povprečna količina zahtevkov označena z modro krivuljo, povprečni odzivni čas označen z zeleno krivuljo in predlagano število vzporednih instanc mikrostoritve označeno z rdečo krivuljo. Na podlagi predlaganega števila instanc mikrostoritve je dodatek Adept Scale samodejno ustvarjal in izklapljal instance mikrostoritve. Iz grafa sta razvidna dva večja sunka zahtevkov, ki ju je možno prepoznati iz vrhov modre krivulje. Ta sunka sta povzročila zvišanje števila instanc



Slika 6.4: Samodejno skaliraje mikrostoritve glede na promet in odzivni čas.

mikrostoritve.

Iz grafa na sliki 6.4 je razvidno spreminjanje predlaganega števila instanc mikrostoritve za delo z izdelki. Iz grafa je razvidno povečanje števila instanc iz ena na tri po prvi večji obremenitviji. V nadaljevanju se število zahtevkov na minuto in predlagano število instanc zniža. Na koncu grafa pa je razvidno še eno povečanje števila instanc mikrostoritve iz dve na štiri ob drugi večji obremenitviji. Na podlagi samodejnega ustvarjanja in izklapljanja instanc mikrostoritev smo uspešno potrdili ključno zahtevo po visoki elastični skalabilnosti arhitekture na samodejen način s pomočjo naprednih funkcij oblačne infrastrukture ponudnika Heroku.

6.3 Verifikacija načrtovalskih vzorcev

Pri implementaciji arhitekture smo upoštevali arhitekturne in načrtovalske vzorce, ki so podrobneje opisani v delu [29]. V tem delu je omenjenih več vzorcev, ki temeljijo predvsem na dveh ključnih vzorcih objektno usmerjenega programiranja. Prvi vzorec govori o programiranju na podlagi vmesnika in ne implementacije, kar omogoča zakrivanje kompleksnosti in implementacije pred odjemalci. Drugi vzorec pa daje prednost kompoziciji razredov pred dedovanjem, ki prinaša izpostavitev razredov implementaciji starševskih razredov. V nadaljevanju tega poglavja so podrobneje opisani posamezni vzorci in kako smo jih upoštevali pri implementaciji arhitekture.

Pri leni inicializaciji zavlačujemo z oblikovanjem objektov, kompleksnimi izračuni ali drugimi obsežnejšimi operacijami dokler niso nujno potrebni, kar lahko pomaga pri odzivnosti in zmanjša potrebni čas zagona storitev. Pred klicem obsežnejše operacije se

preverijo predhodni rezultati operacij in se v primeru odsotnosti tega rezultata izvede klic operacije [29]. Pri implementaciji anotacije za lokacijsko odvisnost smo ta vzorec uporabili pri branju območij delovanja anotacije iz konfiguracijske datoteke, kar je razvidno iz prestreznika anotacije v prilogi B. S tem smo zmanjšali zagonski čas mikrostoritev, kar omogoča hitrejše ustvarjanje instanc pri horizontalnem skaliranju.

Vzorec Proxy predstavlja uporabo vmesnikov za dostop do omrežnih povezav, datotek, obsežnejših objektov v pomnilniku in drugih virov. Na kratko vzorec Proxy omogoča odjemalcem dostop do implementacijskih objektov preko vmesnikov, ki tako skrijejo implementacijske podrobnosti pred odjemalci [29]. Ta vzorec smo uporabili pri izpostavljanju poslovne logike preko vmesnikov API tipa REST, kar je razvidno iz izseka 2.3. Vzorec Proxy smo uporabili pri implementaciji obeh mikrostoritev za delo z izdelki in akcijami, kar omogoča izpostavljanje funkcionalnosti mikrostoritev na enoten način.

Veriga odgovornosti je še eden izmed načrtovalskih vzorcev, ki smo ga upoštevali pri zasnovi anotacije za lokacijsko odvisnost. Veriga odgovornosti predstavlja zaporedje objektov, ki eden za drugim procesirajo objekte, zahtevke ipd. Vsak objekt obdela določen del zahtev, drugo pa posreduje naprej v verigo [29]. V lastni anotaciji smo preverjanje ustreznosti lokacije odjemalca implementirali s prestreznikom anotacije, ki je prikazan v prilogi B. Prestreznik se proži še pred klicem metod, ki so implementirane na virih mikrostoritev. Ustreznost lokacije odjemalca tako povzroči posredovanje zahtevka naprej vzdolž verige odgovornosti, kjer se v nadaljevanju proži poklicana metoda.

Vzorec predlog metod narekuje definicijo ogrodja algoritma v razredu, ki kliče korake oz. metode drugih razredov. Ta način nam omogoča spreminjanje korakov algoritma v podrazredih brez spreminjanja samega ogrodja [29]. V lastni implementaciji anotacije za lokacijsko odvisnost smo to med drugim upoštevali pri izračunu razdalje med dvema lokacijama in izračunu ustreznosti lokacije glede na podano območje. V prestrezniku anotacije za izračun razdalje med dvema lokacijama in izračun ustreznosti lokacije odjemalca glede na podano območje uporabljamo metodo entitete Area, ki nam tako omogoča uporabo različnih algoritmov za izračun razdalj.

Med implementacijo lastne anotacije za lokacijsko odvisnost in obeh mikrostoritev smo upoštevali obilico arhitekturnih in načrtovalskih vzorcev, ki so še dodatno prispevali k celostni verifikaciji arhitekture.

Poglavje 7

Sklepne ugotovitve

Vmesniki API tipa REST izpostavljajo funkcionalnosti mikrostoritev in skupaj z dokumentacijo omogočajo preprosto povezovanje in uporabo teh funkcionalnosti. Lokacijsko odvisni vmesniki API tipa REST pa omogočajo definicijo naprednejših funkcionalnosti za lokacijsko odvisne mikrostoritve. Mikrostoritve so zelo aktualno področje, na kar kaže obilica novosti, člankov in ostalih del [9, 11, 12, 13, 14, 15]. Arhitektura mikrostoritev je v zadnjem času postala izjemno dodelana in trenutno vsebuje ogromno strukturnih elementov, ki jih ni treba več eksplicitno implementirati, temveč so na voljo v sklopu oblačnih infrastruktur PaaS. Sama implementacija mikrostoritev lahko hitro postane kompleksna ob upoštevanju programskih vzorcev in lastnosti mikrostoritev. Pri implementaciji so nam v pomoč ogrodja, kot je recimo ogrodje KumuluzEE, ki omogoča preprosto konfiguracijo brez dodatnega lastnega vključevanja in samodejno celostno združevanje v arhive.

Pred realizacijo glavnih prispevkov smo se v sklopu magistrskega dela lotili poglobljene analize arhitekture mikrostoritev in ostalih sorodnih del. Poiskali in preučili smo sodobne arhitekture mikrostoritev. Poleg tega smo pregledali sorodno literaturo za morebitno že obstoječo zasnovo načina podajanja lokacijske odvisnosti vmesnikov API. Pregledali smo tudi sorodna dela s področja razvoja vmesnikov API tipa REST, ki so nujni za izpostavitev funkcionalnosti mikrostoritev. Na podlagi analize arhitekture mikrostoritev smo v nadaljevanju zasnovali lastno arhitekturo, ki smo jo uporabili pri objavi v oblačno infrastrukturo PaaS ponudnika Heroku. Po zasnovi arhitekture smo se nato lotili opisa zahtev lokacijsko odvisnih vmesnikov API tipa REST in možne definicije le-teh z razširitvijo specifikacije Swagger. Na podlagi nove specifikacije za definicijo lokacijsko odvisnih vmesnikov API tipa REST smo nato implementirali lastno anotacijo za dodajanje lokacijske odvisnosti. V nadaljevanju smo s pomočjo nove anotacije implementirali dve mikrostoritvi in ju konfigurirali z ogrodjem KumuluzEE. Na koncu smo obe mikrostoritvi objavili še v oblačno infrastrukturo in uspešno verificirali oba glavna prispevka magistrske naloge.

Glavni prispevek magistrske naloge je zmožnost definicije lokacijsko odvisnih vmesnikov API tipa REST s specifikacijo Swagger, določanje pravic dostopa odjemalcev in prilagajanje delovanja mikrostoritve glede na lokacijo odjemalca. Poleg same definicije vmesnikov smo v sklopu magistrske naloge implementirali lastno anotacijo na podlagi platforme Java EE in uspešno verificirali njeno delovanje. Ta anotacija omogoča modularen način dodajanja lokacijske odvisnosti razredom, ki implementirajo vmesnike API tipa REST, na uveljavljen način pri implementaciji s platformo Java EE. Lokacijsko odvisnost vmesnikov je tako možno doseči tudi na razredih v drugih projektih z vključitvijo modula Maven in anotiranjem razredov, podobno kot bi to storili za dodajanje varnostnih omejitev dostopa do vmesnikov. Ta način dodajanja lokacijske odvisnosti omogoča preprosto implementacijo lokacijsko odvisnih storitev. Na podlagi nove anotacije smo v nadaljevanju implementirali dve mikrostoritvi, ju konfigurirali z ogrodjem KumuluzEE in verificirali njuno delovanje v oblačni infrastrukturi PaaS.

Ogrodje KumuluzEE je precej novo in zanj obstaja le malo celostnih primerov konfiguracije projektov, ki so implementirani s platformo Java EE. V magistrski nalogi smo v ta namen, ne le podrobno dokumentirali implementacijo mikrostoritve in njeno konfiguracijo z ogrodjem KumuluzEE, temveč tudi dokumentirali objavo in zagon dveh mikrostoritev v oblačni infrastrukturi PaaS priznanega ponudnika Heroku. S tem smo doprinesli k potrditvi konfiguracije z ogrodjem KumuluzEE mikrostoritev, ki so implementirane s platformo Java EE. Pomembnost tega prispevka se kaže tudi v overitvi samodejnega skaliranja mikrostoritev konfiguriranih z ogrodjem KumuluzEE. Na podlagi dokumentacije v tej magistrski nalogi je možno implementirati, konfigurirati in objaviti lastno množico mikrostoritev.

Z uspešno verifikacijo delovanja anotacije za lokacijsko odvisnost in samodejnega skaliranja mikrostoritev v oblačni infrastrukturi smo potrdili implementacijo in zastavljene cilje magistrskega dela. S skladnim delovanjem anotacije z definicijo lokacijsko odvisnih REST vmesnikov API smo potrdili glavni cilj dela in pravilnost implementacije anotacije. S samodejnim skaliranjem mikrostoritev pa smo potrdili pravilnost zasnove arhitekture, implementacije, konfiguracije z orodjem KumuluzEE in objavo v oblačno infrastrukturo Heroku. Osrednja prispevka magistrske naloge sta tako prinesla možnost obsežnejše definicije vmesnikov API tipa REST in potrditev implementacije mikrostoritev s platformo Java EE.

Magistrsko delo ponuja obilico možnosti za nadaljnji razvoj, tako v smeri še naprednejše definicije lokacijsko odvisnih vmesnikov API tipa REST, kot tudi v smeri implementacije ponovno uporabnih modulov ali knjižnic za različne programske jezike. Ena izmed možnosti je tudi razširitev uporabe anotacije za lokacijsko odvisnost, tako da bi delovala tudi na vmesnikih API tipa SOAP. Mikrostoritve navadno izpostavljamo preko vmesnikov API tipa REST, vendar bi funkcionalnost lokacijsko odvisne anotacije lahko dobro izkoristili tudi pri izmenjavi sporočil tipa SOAP.

V sklopu nadaljnjega dela bi lahko še obsežnejše testirali smotrnost dodatnih polj specifikacije Swagger. Specifikacija Swagger omogoča definicijo in opis vmesnikov API tipa REST. Magistrsko delo ponuja predlog razširitve specifikacije z dodajanjem lokacijske odvisnosti. Ob ustreznosti dodatnih polj bi lahko tudi formalno uveljavili razširjeno specifikacijo Swagger z možnostjo lokacijske odvisnosti, določanjem pravic dostopa in spreminjanjem delovanja glede na lokacijo odjemalca.

Razširjena specifikacija Swagger trenutno omogoča definicijo le preprostih območij delovanja. Z zemljepisno dolžino, širino in radijem je mogoče definirati območja, omejena s krogom, ki so dovolj za preproste aplikacije. Za uporabo v poslovnih aplikacijah pa bi potrebovali tudi območja drugih oblik. Območje v obliki pravokotnika bi bilo uporabno za definicijo območij znotraj stavb, kar bi zelo poenostavilo samo definicijo kot tudi izboljšalo natančnost le-te.

Nadgradnjo magistrskega dela bi predstavljala tudi implementacija možnosti dodajanja lokacijske odvisnosti na vmesnike, ki so implementirani z drugimi programskimi jeziki, podobno kot smo to storili z lastno anotacijo v platformi Java EE. Implementacija možnosti za dodajanje lokacijske odvisnosti v drugih jezikih bi močno olajšala razvoj lokacijsko odvisnih mikrostoritev, ki izpostavljajo vmesnike API tipa REST tudi v drugih tehnologijah.

Literatura

[1] J. Schiller, A. Voisard, Location-based services, 1st Edition, Elsevier, 2004.

[2] N. Davies, A. Dey, J. Hightower, E. de Lara, Location-based services, IEEE Pervasive Computing 9 (1) (2010) 11–12.

[3] I. A. Junglas, R. T. Watson, Location-based services, Commun. ACM 51 (3) (2008) 65–69.

[4] X. Chen, Y. Chen, F. Rao, An efficient spatial publish/subscribe system for intelligent location-based services, in: Proceedings of the 2nd international workshop on Distributed event-based systems, DEBS '03, ACM, 2003, pp. 1–6.

[5] F. Alt, A. S. Shirazi, A. Schmidt, U. Kramer, Z. Nawaz, Location-based crowdsourcing: extending crowdsourcing to the real world, in: Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, ACM, 2010, pp. 13–22.

[6] P. T. Eugster, B. Garbinato, A. Holzer, Location-based publish/subscribe, in: Network Computing and Applications, Fourth IEEE International Symposium on, IEEE, 2005, pp. 279–282.

[7] M. Diallo, V. Sourlas, P. Flegkas, S. Fdida, L. Tassiulas, A content-based publish/subscribe framework for large-scale content delivery, Comput. Netw. 57 (4) (2013) 924–943.

[8] T. Hoff, Deep lessons from google and ebay on building ecosystems of microservices (2015).
URL `http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html`

[9] S. Newman, Building Microservices, 1st Edition, O'Reilly Media, Inc., 2015.

[10] S. J. Barbeau, M. A. Labrador, P. L. Winters, R. Pérez, N. L. Georggi, Location API 2.0 for J2ME - A New Standard in Location for Java-enabled Mobile Phones, Vol. 31, Elsevier Science Publishers B. V., 2008.

[11] T. Faganel, M. B. Jurič, Microservices with java ee and kumuluzee (2015).
URL https://blog.kumuluz.com/

[12] N. Kratzke, About microservices, containers and their underestimated impact on network performance, Proceedings of CLOUD COMPUTING 2015 (2015) 165–169.

[13] P. P. Kukade, G. Kale, Auto-scaling of micro-services using containerization, International Journal of Science and Research (IJSR) 4 (9) (2015) 1960–1963.

[14] D. Namiot, M. Sneps-Sneppe, On micro-services architecture, International Journal of Open Information Technologies 2 (9) (2014) 24–27.

[15] C. Richardson, Introduction to microservices (2015).
URL https://www.nginx.com/blog/introduction-to-microservices/

[16] M. Masse, REST API design rulebook, 1st Edition, O'Reilly Media, Inc., 2011.

[17] T. Faganel, Ogrodje za razvoj mikrostoritev v javi in njihovo skaliranje v oblaku, B.Sc. thesis, Univerza v Ljubljani (2015).

[18] J. I. Fernández Villamor, C. A. Iglesias Fernandez, M. Garijo Ayestaran, Microservices: Lightweight service descriptions for rest architectural style, Proceedings of the 2nd International Conference on Agents and Artificial Intelligence, ICAART 2010.

[19] B. Costa, P. F. Pires, F. C. Delicato, P. Merson, Evaluating rest architectures-approach, tooling and guidelines, J. Syst. Softw. 112 (C) (2016) 156–180.

[20] J. Lewis, M. Fowler, Microservices (2014).
URL http://martinfowler.com/articles/microservices.html

[21] C. Richardson, Pattern: Microservices architecture (2014).
URL http://microservices.io/patterns/microservices.html

[22] C. Richardson, Pattern: Monolithic architecture (2014).
URL http://microservices.io/patterns/monolithic.html

[23] T. Erl, Soa: principles of service design, Vol. 1, Prentice Hall Upper Saddle River, 2008.

[24] D. Krafzig, K. Banke, D. Slama, Enterprise SOA: service-oriented architecture best practices, Prentice Hall Professional, 2005.

[25] M. Zur Muehlen, J. V. Nickerson, K. D. Swenson, Developing web services choreography standards—the case of rest vs. soap, Decision Support Systems 40 (1) (2005) 9–29.

[26] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of json and xml data interchange formats: A case study., Caine 2009 (2009) 157–162.

[27] M. L. Abbott, M. T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, 1st Edition, Addison-Wesley Professional, 2009.

[28] G. Lawton, Developing software online with platform-as-a-service technology, Computer 41 (6) (2008) 13–15.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Dodatek A

Definicija vmesnikov API

A.1 Vmesnik API

```
{
 "swagger": "2.0",
 "info": {
  "title": "Izdelki vmesnik API",
  "description": "Vmesnik API za pridobivanje izdelkov.",
  "version": "1.0.0"
 },
 "host": "localhost:3000",
 "schemes": ["http"],
 "basePath": "/v1",
 "produces": ["application/json"],
 "paths": {
  "/izdelki": {
   "get": {
    "summary": "Pridobi izdelke",
    "description": "Metoda za pridobivanje vseh izdelkov.",
    "parameters": [
     {
      "name": "latitude",
      "in": "header",
      "description": "Zemljepisna sirina odjemalca.",
      "required": true,
```

```
      "type": "number",
      "format": "double"
     },
     {
      "name": "longitude",
      "in": "header",
      "description": "Zemljepisna dolzina odjemalca.",
      "required": true,
      "type": "number",
      "format": "double"
     }
    ],
    "tags": ["Izdelki"],
    "responses": {
     "200": {
      "description": "Seznam izdelkov",
      "schema": {
       "type": "array",
       "items": {"$ref": "#/definitions/Izdelek"}
      }
     },
     "default": {
      "description": "Napaka",
      "schema": {"type": "string"}
     }
    }
   }
  }
 },
 "definitions": {
  "Izdelek": {
   "type": "object",
   "properties": {
    "id": {
     "type": "integer",
     "format": "int32",
     "description": "ID izdelka."
    },
```

```
    "naziv": {
     "type": "string",
     "description": "Ime oz. naziv izdelka."
    },
    "opis": {
     "type": "string",
     "description": "Podroben opis izdelka"
    },
    "cena": {
     "type": "number",
     "format": "double",
     "description": "Cena izdelka."
    }
   }
  }
 }
}
```

Izsek A.1: Vmesnik API tipa REST definiran s specifikacijo Swagger.

A.2 Lokacijsko odvisni vmesnik API

```
{
 "swagger": "2.0",
 "info": {
  "title": "Izdelki vmesnik API",
  "description": "Vmesnik API za pridobivanje izdelkov.",
  "version": "1.0.0"
 },
 "host": "localhost:3000",
 "schemes": ["http"],
 "basePath": "/v1",
 "produces": ["application/json"],
 "locationDefinitions": {
  "center": {
   "type": "circle",
   "description": "Center delovanja vmesnika API.",
   "latitude": 46.077944,
```

```
   "longitude": 14.500017,
   "radius": 100
  },
  "obrobje": {
   "type": "circle",
   "description": "Obrobje delovanja vmesnika API.",
   "latitude": 46.077944,
   "longitude": 14.500017,
   "radius": 500
  }
 },
 "paths": {
  "/izdelki": {
   "get": {
    "summary": "Pridobi izdelke",
    "description": "Metoda za pridobivanje vseh izdelkov.",
    "parameters": [
     {
      "name": "latitude",
      "in": "header",
      "description": "Zemljepisna sirina odjemalca.",
      "required": true,
      "type": "number",
      "format": "double"
     },
     {
      "name": "longitude",
      "in": "header",
      "description": "Zemljepisna dolzina odjemalca.",
      "required": true,
      "type": "number",
      "format": "double"
     }
    ],
    "onLocation": [
     "center",
     "obrobje"
    ],
```

```
    "tags": ["Izdelki"],
    "responses": {
     "200": {
      "description": "Seznam izdelkov",
      "schema": {
       "type": "array",
       "items": {"$ref": "#/definitions/Izdelek"}
      },
      "onLocation": [
       "center"
      ]
     },
     "default": {
      "description": "Napaka",
      "schema": {"type": "string"}
     }
    }
   }
  }
},
"definitions": {
 "Izdelek": {
  "type": "object",
  "properties": {
   "id": {
    "type": "integer",
    "format": "int32",
    "description": "ID izdelka."
   },
   "naziv": {
    "type": "string",
    "description": "Ime oz. naziv izdelka."
   },
   "opis": {
    "type": "string",
    "description": "Podroben opis izdelka"
   },
   "cena": {
```

```
      "type": "number",
      "format": "double",
      "description": "Cena izdelka."
     }
    }
   }
  }
}
```

Izsek A.2: Lokacijsko odvisni vmesnik API tipa REST definiran z razširjeno specifikacijo Swagger.

Dodatek B

Prestreznik anotacije

```
@OnLocation
@Interceptor
public class LocationInterceptor {

  @Inject
  private HttpServletRequest request;

  @AroundInvoke
  public Object checkLocation(InvocationContext context) {
    Location clientLocation;
    if (request.getHeader("latitude") == null || request.
        getHeader("longitude") == null)
      return Response.status(Status.BAD_REQUEST).build();
    else {
      try {
        clientLocation = new Location(Double.parseDouble(request
            .getHeader("latitude")), Double.parseDouble(request.
            getHeader("longitude")));
      } catch (Exception e) {
        return Response.status(Status.BAD_REQUEST).build();
      }
    }

    OnLocation onLocation = getOnLocation(context);
```

```
    // Preveri ustreznost lokacije glede na podano obmocje
    if (onLocation != null && onLocation.latitude() != 0 &&
        onLocation.longitude() != 0) {
      Area area = new Area(onLocation.latitude(), onLocation.
          longitude(), onLocation.radius());
      if (area.contains(clientLocation))
        return context.proceed();
      else
        return Response.status(Status.FORBIDDEN).build();
    }

    List<Area> areas = parseAreaList();
    // Preveri ustreznost lokacije glede na podano ime
    if (areas != null && onLocation != null && !onLocation.name
        ().isEmpty()) {
      for (Area a : areas) {
        if (a.getName() != null && a.getName().equals(onLocation
            .name()) && a.contains(clientLocation))
          return context.proceed();
      }
    }
    // Preveri ustreznost odjemalceve lokacije glede na obmocja
    if (areas != null && onLocation != null && onLocation.name()
        .isEmpty()) {
      for (Area a : areas) {
        if (a.getName() != null && a.contains(clientLocation)) {
          request.setAttribute("area", a.getName());
          return context.proceed();
        }
      }
    }

    return Response.status(Status.FORBIDDEN).build();
  }
  ...
}
```

Izsek B.1: Prestreznik anotacije za lokacijsko odvisnost.